EVALUATING THE EFFECTIVENESS AND EFFICIENCY OF PARSONS PROBLEMS AND DYNAMICALLY ADAPTIVE PARSONS PROBLEMS AS A TYPE OF LOW COGNITIVE LOAD PRACTICE PROBLEM

A Dissertation
Presented to
The Academic Faculty

by

Barbara Jane Ericson

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Human Centered Computing

School of Interactive Computing
Georgia Institute of Technology
May 2018

EVALUATING THE EFFECTIVENESS AND EFFICIENCY OF PARSONS PROBLEMS AND DYNAMICALLY ADAPTIVE PARSONS PROBLEMS AS A TYPE OF LOW COGNITIVE LOAD PRACTICE PROBLEM

Approved by:

Dr. James D. Foley, Advisor
School of Interactive Computing
Georgia Institute of Technology

Dr. Richard Catrambone
School of Psychology
Georgia Institute of Technology

Dr. Amy S. Bruckman
School of Interactive Computing
Georgia Institute of Technology

Dr. Alan C. Kay
Computer Science Department
University of California, Los Angeles

Dr. Ashok K. Goel
School of Interactive Computing
Georgia Institute of Technology

Dr. Mitchel Resnick
Media Laboratory
Massachusetts Institute of Technology

Date Approved: March 12, 2018

To my grandmother, Opal Peters Hund, who always wished that she could have continued her education. To my mother, Janet Hund Ericson, who showed me that you *can* go back to school, even if you are older than most students and still have children at home. She earned a nursing degree in 1975 when I was a teenager.

ACKNOWLEDGEMENTS

I thank my husband for beliving in me and supporting me while I worked full-time and pursued my PhD part-time. I also thank my children, Matthew, Katherine, and Jennifer for their love, for keeping me grounded, and for helping to lessen my load in the final sprint to the end. I thank Rosa Arriaga for pushing me to go for my PhD and for refusing to let any of my concerns stop me from applying. I thank Lauren Margulieux for running statistical tests in SPSS. I thank Matthew Guzdial for running statistical tests in R. I thank Katherine Guzdial for reviewing the statistical tests and results. I thank the whole js-Parsons team for creating the original open-source Parsons software and sharing it on github (Ville Karavirta, Petri Ihantola, Juha Helminen). I thank Mike Hewner for porting js-Parsons to the Runestone platform. I thank Jochen (Jeff) Rick for improving the original js-Parsons code and user interface. I thank the leaders and attendees of the ICER Doctoral Consortium for helping me refine my studies. I especially thank my advisor, Jim Foley, for being a terrific model of how to be both a good person and great professor.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

AP	Advanced Placement – a secondary course that offers college credit and or placement based on the score on an exam
CSA	The AP Computer Science A course – equivalent to a first college course for majors
CSP	The AP Computer Science Principles course – equivalent to a first college course for non-majors
ITS	Intelligent Tutoring System – a software system tries to achieve similar learning gains to a human tutor

SUMMARY

Many countries, including the United States, want to provide access to computer science to all secondary students. To accomplish this goal, thousands of secondary teachers with no programming experience need to learn programming. However, learning to program can be difficult. Novice programmers spend hours trying to fix errors in their programs, like unmatched parentheses. Most introductory programming courses require novices to learn by writing many small programs. However, writing programs, even short ones, is a complex cognitive task, which can easily overwhelm novices and impede learning. Busy secondary teachers need a more efficient way to learn programming.

One way to potentially decrease the difficulty of learning to program is to use Parsons problems to give novices practice with syntactic and semantic errors as well as common algorithms. *Parsons problems* are a type of code completion problem in which the correct code is provided, but is broken into code blocks that are mixed up. The learner must place the blocks in the correct order. A *two-dimensional Parsons problem* also requires the code blocks to be indented to show the structure of the solution (such as indicating blocks that belong in the body of a loop). Parsons problems can contain *distractor* blocks, that should not be used in a solution because they contain syntactic or semantic errors. Distractor blocks can be randomly mixed in with the correct blocks, called *un-paired distractors,* or distractor blocks can be shown directly above or below the corresponding correct code block, called *paired distractors*.

I initially added 11 Parsons problems without distractors to a chapter of a free interactive electronic book (ebook) and observed teachers as they worked through that chapter. The teachers found the problems interesting, and were able to solve the problems in one or two attempts. A log file analysis of student use of that same ebook showed that most students solved the Parsons problems in a couple of attempts, however some students took over 20 attempts to solve a problem, and some students never solved them. More students attempted to solve Parsons problems than nearby multiple-choice questions, which suggests that students find Parsons problems engaging.

To test the efficiency (time to complete the instructional practice problems) and effectiveness (learning gain from pretest to posttests) of learning with Parsons problems, I conducted a between-subjects study with three conditions: 1) solving Parsons problems with distractors containing syntactic and semantic errors, 2) fixing code with the same syntactic and semantic errors, or 3) writing the equivalent code. The students in the Parsons condition took significantly less time than the students in either the fix or write conditions to complete the instructional problems. While there was a significant gain for all three conditions from pretest to immediate posttest and to the delayed posttest, there was no significant difference by condition. This implies that students in the Parsons problem condition learned as much as students in the fix code or write code condition. This study provided initial evidence that solving Parsons problems could be a more efficient, but just as effective, form of practice than having students write code or fix code.

While teachers found the Parsons problems in my observations interesting, some wanted them to be harder. Yet, some students struggled to solve the exact same problems.

To satisfy the advanced learners' desire for greater difficulty and the novices need for additional help, I added intra-problem adaptation. This means that if the learner is struggling to solve the current Parsons problem and asks for help, the problem is dynamically made easier by disabling distractors, providing indentation, or combining blocks. The goal is to keep the learner in the *zone of proximal development,* which means that the learner can solve the problem with help, but not unaided.

To test intra-problem adaptation, I conducted an observational study with 11 teachers as they solved both intra-problem adaptive and non-adaptive Parsons problems, as well as fix code and write code problems. The teacher had to click the *Help Me* button to initiate the intra-problem adaptation, which would disable a distractor, provide indentation, or combine two blocks into one. Adaptation was not available until at least three *full* solutions had been attempted. A full solution is one with at least the same number of blocks as the correct solution. Most teachers preferred the intra-problem adaptive Parsons problems to the non-adaptive Parsons problems, though some teachers were concerned that students might overuse the help. Teachers understood most of the intra-problem adaptation process. They were confused when distractors were disabled that they hadn't used in their solution, and they didn't always understand the implicit hints when the indentation was provided. All 11 teachers were able to solve all of the adaptive Parsons problems, but two teachers gave up before solving a non-adaptive Parsons problem.

To test the intra-problem adaptive Parsons problems in actual use, I incorporated the same problems into an ebook. Log file analysis of student use of this ebook provided evidence that a higher percentage of students who used the intra-problem adaptation got

the problem correct than students who didn't use it. However, this result may be because some students gave up on solving the problem before the adaptation was enabled (after at least three full attempts).

Finally, I tested the effectiveness and efficiency of learning from solving adaptive (both intra-problem and inter-problem) Parsons problems by conducting a between-subjects study with four conditions: 1) solving adaptive Parsons problems with distractors, 2) solving non-adaptive Parsons problems with distractors, 3) writing the equivalent code, and 4) a control group that solved off-task adaptive Parsons problems. Inter-problem adaptation adjusts the difficulty of the next problem based on the number of attempts needed to solve the previous problem. If the learner struggled to solve the previous problem, the next problem is made easier, and if the learner solved the previous problem in just one attempt, the next problem is made harder. Problems can be made harder by un-pairing distractors so that they are randomly mixed in with the correct code or by adding more distractors. Problems can be made easier by pairing the distractor and correct code blocks or reducing the number of distractors.

The students in both of the on-task Parsons conditions, adaptive and not-adaptive, took significantly less time than the students in the write code condition to complete the four instructional problems. This result provides additional evidence of the efficiency of solving Parsons problems versus writing the equivalent code. While there was a significant gain for all three conditions from pretest to immediate posttest and to the delayed posttest, there was no significant difference by condition. This result supports the hypothesis that solving Parsons problems can be as effective for learning as writing code. There was also a significant difference for the scores from the pretest to the immediate

posttest between the control group (the off-task adaptive Parsons) and the on-task adaptive Parsons condition, which provides evidence of learning from solving adaptive Parsons problems. However, there was no significant difference between the control group and the non-adaptive Parsons group or the write code group, which implies that at least some of the learning gains were from repeated exposure to the same or similar problems with feedback.

This work contributes to the research on Parsons problems and adaptive learning. It includes the first study to compare the effectiveness and efficiency of solving Parsons problems with distractors, fixing code with the same errors, and writing the equivalent code. I invented two types of adaptation for Parsons problems: intra-problem and inter-problem. Intra-problem adaptation is similar to the hints and feedback provided by Intelligent Tutoring Systems as the learner solves a problem (ITS), but the hints are implicit rather than explicit and are only given after a complete solution has been constructed, rather than at each step. This allows for desirable difficulties which should improve long-term retention. Inter-problem adaptation is similar to the selection process in an ITS, in that the difficulty of the next problem is based on the learner's performance on the prior problem. However, rather that selecting a problem based on the learner's performance, I modify the difficulty of the next problem.

I tested the impact of adaptable Parsons problems with both qualitative and quantitative studies. These studies provided evidence that adaptable Parsons problems are a more efficient and just as effective form of practice than writing code from scratch, though further studies should be done to strengthen the evidence. The studies demonstrated that most learners preferred adaptive Parson problems to non-adaptive

ones, correctly completed more problems if they used the adaptation, and perceived that Parsons problems helped them learn to fix and write code.

This work also contributes software to facilitate the use and study of Parsons problems. Our research group added the open-source js-parsons software to the Runestone Interactive ebook platform. We improved the user interface by adding guidelines to signify that indentation was possible, providing support for touch screens, and improving the feedback for incorrect indentation. I led the effort to add intra-problem and inter-problem adaptation, which should help more students successfully solve and learn from Parsons problems. My research group has authored hundreds of Parsons problems in several free interactive ebooks, which are already being used by tens of thousands of students. Researchers can leverage this work to continue to test the effectiveness and efficiency of solving Parsons problems, and instructors can use this work to provide learners with an engaging and effective form of practice.

If solving Parsons problems with distractors helps novices to learn as well as fixing or writing code, they could be used to provide learners with a more efficient and just as effective form of practice. This could reduce the frustration that many novices feel when learning programming and perhaps improve the retention rates in college-level introductory programming courses. Interactive ebooks, with adaptive Parsons problems, could help prepare thousands of secondary teachers to teach introductory computing courses by providing efficient programming practice.

CHAPTER 1. INTRODUCTION & MOTIVATION

The National Science Foundation (NSF) has been trying to increase the number of secondary computing teachers in the United States in order to provide more access to computing education and to increase the quantity and diversity of students studying computing at the college level. This effort was originally called the CS10K effort since it tried to prepare 10,000 secondary teachers to teach introductory computer science courses by the fall of 2016 (Astrachan, Cuny, Stephenson, & Wilson, 2011). The National Science Foundation is also concerned about the underrepresentation of women and minorities in computing, so it partnered with the College Board to create a new Advanced Placement (AP) course, Computer Science Principles (CSP), which was piloted at many colleges/universities and high schools in the United States. It was first offered as an Advanced Placement course during the 2016-2017 academic year. The goal was to have at least 20,000 students take the AP CSP exam in the first year. Instead, over 40,000 students took the AP CSP exam in the United States. While this was the largest number of students ever to take a new AP exam, we still have a long way to go to reach the level of AP U. S. History, which had over 500,000 exam takers in 2017.

One challenge in trying to reach 500,000 AP CSP exam takers, is the need to prepare thousands more high school teachers to teach the course. Most teachers have not had any programming experience. While we have had success with in-person professional development of teachers (Bruckman et al., 2009), this type of professional development is expensive. Teachers without programming experience are often not

confident in their ability to teach programming after a one-week in-person professional development workshop (B. Ericson, Guzdial, & Biggers, 2005). This is not surprising since spaced practice is better than massed practice for long-term retention of information (R. A. Bjork, Dunlosky, & Kornell, 2013). It is likely that preparing thousands of teachers to teach the programming part of the new CS Principles course will require at least some on-line self-paced learning that encourages spaced practice. The CSLearning4U research group that I have worked with over the last six years believes that an on-line ebook (electronic book) with worked examples and low cognitive load practice problems such as multiple-choice questions, fill in the blank questions, and Parsons problems can provide spaced practice to allow in-service teachers to learn introductory programming concepts efficiently and effectively. The goal of the CSLearning4U project is to provide effective learning opportunities that fit into teachers' lives, e.g., the teachers can reliably predict the amount of time a lesson will take (B. Ericson, Moore, Morrison, & Guzdial, 2015).

One barrier to preparing more computing teachers is the difficulty of learning to program. Students have spent many frustrating hours trying to figure out why their program doesn't compile or doesn't produce the expected output (Benda, Bruckman, & Guzdial, 2012). Drop out and failure rates in many introductory computing classes at the college level are high with an average pass rate worldwide of only 67% (Bennedsen & Caspersen, 2007; Watson & Li, 2014). College students that encounter errors while programming experience negative emotions that impact self-efficacy (Kinnunen & Simon, 2010). Negative experiences in courses tend to affect women more than men

(Dweck, 1986; Margolis & Fisher, 2003) which may be one reason that women are underrepresented in computing.

Beginning programming students have to learn many things. They have to develop a mental model of the computer (the notational machine), the notation (syntax and semantics), the structures (schemas), and develop skill in planning, developing, and debugging programs (Boulay, 1988). Piaget popularized the term *schema*, which is a mental framework for organizing and applying knowledge (Wadsworth, 1989). Experts have a large number of schemas that they can use to recognize and solve similar problems (Atkinson, Derry, Renkl, & Wortham, 2000; Winslow, 1996). It can take 10 years of experience to turn a novice programmer into an expert programmer (Winslow, 1996). In the ACT family of cognitive models created by Anderson, abstract knowledge like schemas can only be learned through practice (John R. Anderson, 1983; John R. Anderson, Bothell, & Byrne, 2004). Anderson created several *Intelligent Tutoring Systems (ITS)* to test his ACT models (J. R. Anderson, Boyle, Corbett, & Lewis, 1990). These systems select practice problems based on the learner's performance and try to keep the learner on the optimal path to a solution through hints. ITS have sometimes achieved a standard deviation better results than those who learned without the tutors, but not always (John R. Anderson, Corbett, Koedinger, & Pelletier, 1995). Additionally, these systems take a good bit of time and expertise to create. It can take 23 hours to create one hour of instruction for a simple ITS and many require 100 hours or more (Corbett, Koedinger, & Anderson, 1997; Folsom-Kovarik, Schatz, & Nicholson, 2010).

Even through ITS have had positive results, especially with trained teachers, they haven't been widely used (John R. Anderson et al., 1995).

In introductory programming courses at the college level, students are mostly expected to practice by writing code. Writing code can take a large and unpredictable amount of time. Students have reported spending hours trying to fix a simple syntax error like a comma out of place (Benda et al., 2012). Writing code is an *authentic task* in an introductory programming course. An *authentic task* is one that someone in the field of study might encounter in their work (Shaffer & Resnick, 1999). Constructivists encourage the use of authentic tasks to motivate students. However students can easily be overwhelmed by the complexity of an authentic task (v. Merriënboer, Kirschner, & Kester, 2003). *Cognitive load theory (CLT)* states that the human mind has limited processing capability and that the cognitive load of some complex tasks must be reduced in order for learning to occur (Sweller, 2010).

1.1 Reducing Cognitive Load with Parsons Problems

One of the recommended approaches to reducing cognitive load is to use completion tasks rather than whole tasks (J. J. G. V. Merriënboer, 1990; J. J. G. V. Merriënboer & Croock, 1992). An example of a completion task is the modification or extension of a program rather than writing a program from scratch. *Parsons problems* are a type of code completion practice problem in which the learner must place blocks of mixed up program code into the correct order. Each block can contain one or more program statements as shown in Figure 1. The code on the left is mixed up and the code on the right has been placed in the correct order.

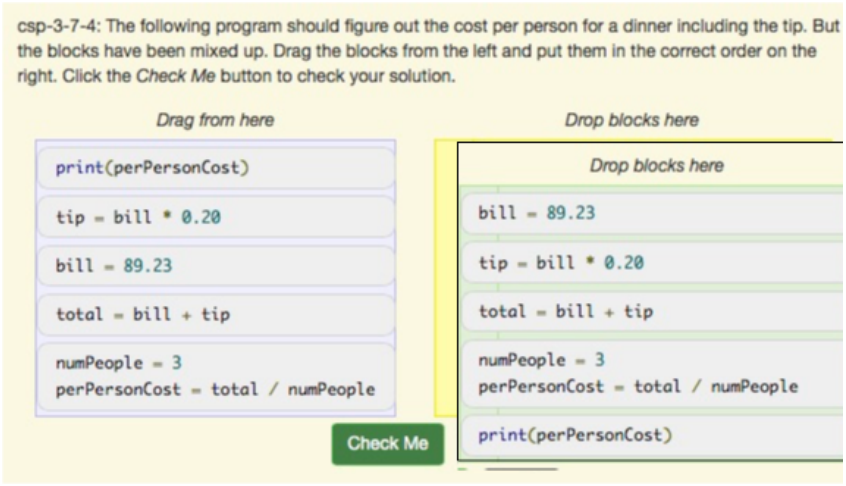


Figure 1. Initially mixed up code on the left and the solution on the right.

Parsons problems can have *distractor* code blocks that are not needed in the correct solution. The distractor blocks can include syntactic errors like the wrong case for a variable name as shown in Figure 2 as well as semantic errors like the wrong boundary condition on a loop.

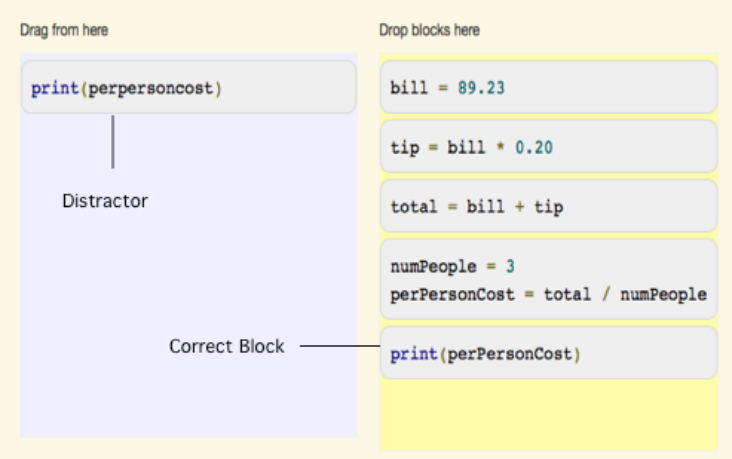


Figure 2. A distractor code block on the left (wrong case on the variable name)

Parsons problems can be used to teach syntactic and semantic language constructs as well as expose students to common programming plans (Parsons & Haden, 2006). Several researchers have hypothesized that solving Parsons problems should result in

more effective and efficient learning than having students write the equivalent code, but to my knowledge none have empirically tested that hypothesis.

1.2 Justification for Exploring Parsons Problems

Our research group at Georgia Tech has been creating ebooks that incorporate a worked-examples plus practice approach from educational psychology (Sweller & Cooper, 1985) to try to make learning more efficient and effective, especially for time-strapped in-service teachers (B. Ericson, S. Moore, et al., 2015). This approach should help in the effort to prepare more secondary computing teachers. It may also help high school and undergraduate students learn to program more effectively and efficiently.

As part of this effort, I have been creating Parsons problems as a type of low cognitive load practice problem. As discussed in chapter 3, I conducted an observational study of teachers working through a chapter of an ebook that contained Parsons problems. The teachers found the problems interesting, and felt that they helped them learn the typical order for the statements, but felt that the problems were too easy. I had intended the Parsons problems for teachers who didn't have any prior programming experience, but all the teachers in my study had been teaching blocks-based programming for over a year. One teacher had already completed a college-level programming course.

In contrast, in my log file analysis of students solving Parsons problems, I found that some novice students clearly struggled to solve the same Parsons problems as the teachers. While most students were able to solve most of the problems in a few attempts, it took some students a surprising number of attempts (e.g., over a hundred attempts) to solve the problems and some students gave up and never solved them. In the prior research on Parsons problems, researchers had experimented with several different types of Parsons problems and found that some types were more difficult than others. Since the teachers in my observational study found the Parsons problems too easy, but the log file

analysis showed that at least some students had great difficulty with the Parsons problems, I decided to investigate the effect of dynamically adaptive Parsons problems.

In a dynamically adaptive Parsons problem the difficulty of the problem changes based on the learner's prior performance on a previous Parsons problem as well as performance on the current problem. For example, if the learner has solved a prior problem in just one attempt, then the difficulty of the next problem will be increased in order to make sure that the learner experiences *desirable difficulties*, which lead to improved long-term learning (E. L. Bjork & Bjork, 2011). This idea of *desirable difficulty* is also consistent with Vygotsky's *zone of proximal development* (ZPD) (Berk & Winsler, 1995) and Ericsson's notion of *deliberate practice* (K. Anders Ericsson, 2006). These concepts will be discussed further in chapter 2.

Based on the results from prior research, Parsons problems can be made more difficult by requiring the solver to provide the structure of the solution (indention or curly braces to indicate block structure), increasing the number of distractors, and/or using un-paired distractors versus paired distractors (Denny, Luxton-Reilly, & Simon, 2008). In paired distractors, the distractor block is randomly shown either above or below the correct code block as shown on the left in Figure 3. In un-paired distractors, the distractor blocks are randomly mixed in with the correct code blocks as shown on the right in Figure 3.

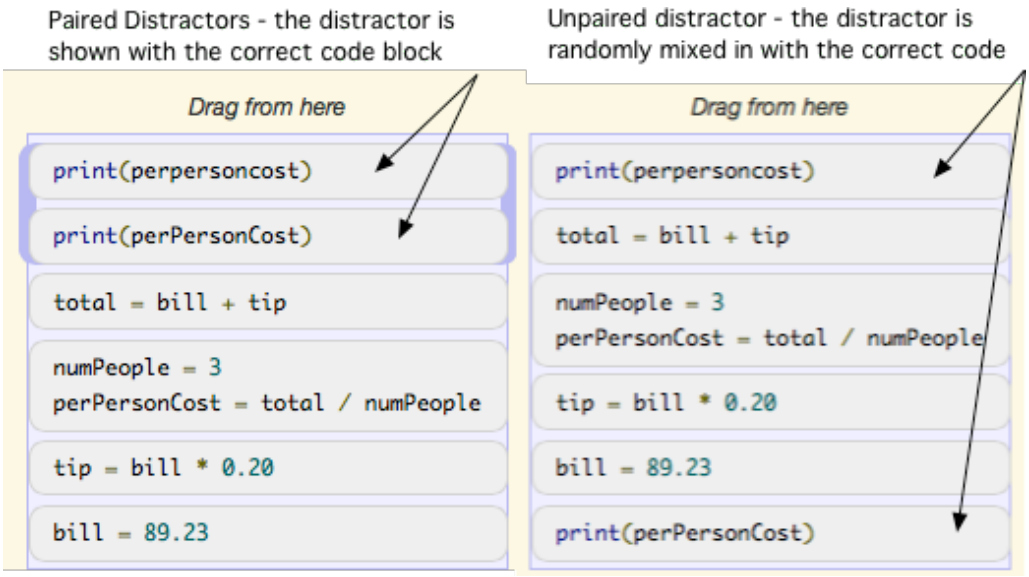


Figure 3. A Parsons problem with a paired distractor on the left and an un-paired distractor on the right

In a dynamically adaptive Parsons problem, if the learner solves a problem in the first attempt, then the next problem will be made more difficult. I call this inter-problem adaptation. Conversely, if the learner is having difficulty solving the current problem as indicated by the number of incorrect attempts, then the current problem can be modified to be easier by reducing the number of distractors, pairing distractors and correct code, providing the structure of the solution, and/or reducing the number of code blocks. I call this intra-problem adaptation.

1.3 Statement of the Problem

We need to prepare thousands of high school teachers in the United States, who do not have any prior programming experience, to teach introductory programming concepts. Prior research has shown that learning to program is difficult and time consuming. The difficulty of learning to program negatively impacts learners and leads

to a higher percentage of women leaving the major than men. Programming is typically taught by requiring learners to write lots of code. While this is an authentic task, it can lead to cognitive overload, which impedes learning and leads to frustration and decreases self-efficacy.

1.4 My Solution: Dynamically Adaptive Parsons Problems

To increase the effectiveness and efficiency of learning to program, especially for time-strapped in-service teachers, we should use lower-cognitive load practice problems such as Parsons problems, rather than require learners to mostly write code from scratch. Based on cognitive load theory and prior research I predicted that solving Parsons problems with distractors that contain syntactic and semantic errors would lead to at least the same learning and retention of programming knowledge than fixing the same code with syntactic and semantic errors, or than writing the equivalent code. This would mean that solving Parsons problems would lead to just as effective learning as writing code from scratch or fixing code with errors. If learners were able to solve Parsons problems significantly faster than writing the equivalent code, then solving Parsons problems would be a more efficient form of learning than writing or fixing code.

While Parsons problems may be a more efficient and effective way to learn compared to fixing or writing code, they could still be improved. I created dynamically adaptive Parsons problems where the difficulty of the problem is based on the learner's past and current performance. Based on the theories of desirable difficulties, the zone of proximal development, and deliberate practice, I expected that dynamically adaptive Parsons problems would lead to more efficient and effective learning than the current

non-adaptive Parsons problems. Since some users find Parsons problems too easy and others find them difficult, I expected that learners would prefer dynamically adaptive Parsons problems to non-adaptive Parsons problems. The goal is to keep the learner in Vygotsky's zone of proximal development, so that she is challenged, but not frustrated.

Thesis Statement

Dynamically adaptive Parsons problems will lead to as effective and more efficient learning than fixing code, writing code, or non-adaptive Parsons problems. This will be due to reduced cognitive load, desirable difficulty, and deliberate practice in the learner's zone of proximal development. Leaners will prefer dynamically adaptive Parsons problems versus non-adaptive Parsons problems.

1.5 Research Questions and Hypotheses

RQ1: What is the efficiency (time to complete practice problems), effectiveness (learning gains from pretest to posttests), and cognitive load of 1) solving non-adaptive Parsons problems with distractors versus 2) fixing the same code with the same errors as the distractors versus 3) writing the equivalent code?

I had three hypotheses related to this research question.

- **H1A:** Learners who solve Parsons problems will finish the instructional problems faster than the learners who fix or write code.
- **H1B:** Learners who solve non-adaptive Parsons problems with distractors will achieve similar learning gains from pretest to immediate posttest and delayed posttest than learners who fix code with the same errors as the distractors or learners who write the equivalent code.
- **H1C:** Learners will report lower cognitive load after solving Parsons problems versus learners who write or fix code.

RQ2: Will learners understand the intra-problem adaptation process?

H2A: Most learners will understand the intra-problem adaptation process.

RQ3: What is the effect on correct completion and preference from solving 1) intra-problem adaptive Parsons problems versus 2) non-adaptive Parsons problems?

I have two hypotheses related to this research question.

- **H3A:** A greater percentage of learners will correctly complete intra-problem adaptive Parsons problems than will complete non-adaptive Parsons problems.
- **H3B:** Most learners will prefer adaptive Parsons problems to non-adaptive Parsons problems.

RQ4: Will learners perceive that solving Parsons problems with distractors helped them learn to fix code with similar errors and write similar code?

H4A: Most learners will perceive that solving Parsons problems with distractors helped them learn to fix code with similar errors and write similar code.

RQ5: What is the efficiency (with respect to completion time) and effectiveness (with respect to learning gains from pretest to posttests) of 1) solving adaptive (both intra-problem and inter-problem) Parsons problems versus 2) solving non-adaptive Parsons problems versus 3) writing the equivalent code versus 4) solving off-task adaptive Parsons problems.

I have 3 hypotheses related to this research question.

- **H5A:** Learners who solve on-task (related to the pretest questions) adaptive and non-adaptive Parsons condition will finish the instructional problems significantly faster than the learners who write code.
- **H5B:** Learners who solve on-task adaptive Parsons problems with distractors will achieve similar learning gains from pretest to posttest than learners who solve on-task non-adaptive Parsons problems or learners who write the equivalent code.
- **H5C:** Learners who solve off-task (not related to the pretest questions) adaptive Parsons problems (the control group) will have lower learning gains than those who solve on-task problems.

Originally, I had also intended to test the self-reported cognitive load in **RQ5** as well, but after the first between-subjects study didn't find any significant difference in cognitive load between conditions, I dropped the self-reported cognitive load from the research question.

A summary of the studies, the research questions and hypotheses, and the chapter that each is described in is shown in Table 1.

Table 1. Summary of studies, research questions, and hypotheses

Study	Research Questions and Hypotheses	Chapter
Between-subjects study with three conditions: Parsons, fix, or write code	**RQ1:** efficiency, effectiveness, and cognitive load of Parsons, fix code, and write code? **H1A:** Parsons will be faster to complete **H1B:** Parsons will have similar learning gains from pretest to posttest to the other conditions **H1C:** Parsons will have less cognitive load	4
Observational study of teachers solving intra-problem adaptive and non-adaptive Parsons problems as well as fixing and writing code in an ebook	**RQ2:** understandability of intra-problem adaptation? **H2A:** most will understand **RQ3:** effect on completion and preference? **H3A:** more will correctly complete adaptive **H3B:** most will prefer adaptive **RQ4:** perception of usefulness of Parsons in learning to fix and write code **H4A:** most will find Parsons useful in learning to fix and write code	5
Log file analysis of students solving intra-problem and inter-problem adaptive and non-adaptive Parsons problems in an ebook	**RQ3:** effect on completion? **H3A:** more will correctly complete adaptive Parsons problems	5
Between-subjects study with four conditions: on-task adaptive Parsons, on-task non-adaptive Parsons, write code, and off-task adaptive Parsons (turtle graphics)	**RQ5:** efficiency and effectiveness of intra-problem and inter-problem adaptive Parsons, non-adaptive Parsons, write code, and a control group that solved off-task adaptive Parsons problems on turtle graphics? **H5A:** Parsons will be faster to complete **H5B:** Equivalent learning gains for all on-task groups **H5C:** Less learning gains for control group	6

These research questions and hypotheses are based on prior research in several areas, which is discussed in chapter two. They arose from my initial studies of Parsons

problems, as described in chapter three. The hypotheses were tested by three more studies, which are described in chapters four through six. I summarize the studies and their contributions in chapter 7 as well as my short-term and long-term future work.

1.6 Purpose of Studies & Contributions

Several researchers had hypothesized that Parsons problems could lead to more efficient and effective learning than writing the equivalent code. However, this question had not been empirically tested. I compared solving Parsons problems with distractors that contained syntactic and semantic errors, to fixing the same code with the same errors, to writing the equivalent code. If solving Parsons problems leads to more efficient (as measured by time to complete practice problems) and effective learning (as measured by performance gains from pretest to posttest) than fixing code or writing code, it has major implications for how we teach introductory programming. Teaching programming with an effective, but lower-cognitive load activity could help the United States develop more high school teachers who feel confident in their ability to teach introductory programming concepts with less effort than taking traditional classes focusing on code-writing. The use of Parsons problems could potentially lead to more efficient and effective learning in all introductory programming courses.

I led the effort to modify the open source tool, js-parsons, to support dynamically adaptive Parsons problems. No other current system supports dynamically adaptive Parsons problems in which the current problem can dynamically be made easier if the user is struggling to solve it (intra-problem adaptation) or the next problem can be made easier or harder depending on the user's performance on the previous problems (inter-

problem adaptation). This modified js-parsons code is freely available, which means that other researchers can easily use and evaluate this new type of Parsons problem. In addition, I and the undergraduate and high school students that work with me have created hundreds of Parsons problems that can be used by others, either by using the publically available ebooks that contain them, or by creating their own ebooks.

CHAPTER 2. GROUNDING THE RESEARCH

In this research, I am drawing from work in studies of novice programmers, how people learn, cognitive science, educational psychology, worked examples, intelligent tutoring systems, data-driven hint generation, and Parsons problems.

2.1 Multi-Institution Studies of Novice Programmers

A series of medium to large-scale multi-national and multi-institutional studies of novice programmers have looked at students' ability to write code, read and comprehend code, and trace code. The results show that students perform poorly on tests of introductory concepts and knowledge.

2.1.1 McCracken Working Group Study (MWG) – ITiCSE

The first large-scale multi-national and multi-institutional study has been called the McCracken Working Group (MWG) study (McCracken et al., 2001). It found that undergraduate students who had completed one or two courses in computer science scored much lower than their instructors expected on lab-based code-writing problems to evaluate postfix and infix expressions. The average score for how similar the student solution was to a correct solution was 2.3 (46%) out of 5 points.

However, many students, especially those who didn't do well on the problems, ranked the problems as difficult, hard, or impossible. Students also complained that they didn't have enough time to complete the problems. As the authors admit, the first problem was very difficult for students who didn't have a good understanding of stacks (like a stack of plates in a cafeteria where you can add to the top and remove from the

top) and other data structures. The results showed that a large group of students did very poorly and a smaller group of students had somewhat better performance.

2.1.2 Leeds Working Group Study (LWG)

The Leeds Working Group (LWG) study (Lister et al., 2004) administered 12 multiple-choice questions about array processing to over 500 undergraduate students to test their ability to read and comprehend code. Most of the students had completed or were about to complete their first computer science course. Seven (58%) of the questions required the student to select the correct value of a variable after code executed and five (42%) questions required the student to pick the correct code to complete the program. As the group expected, it found better results that the McCracken Working Group, since the questions were about code comprehension rather than code writing and testing: 51% percent of the students scored 8 (66%) or above out of a possible score of 12. Students who drew pictures, did calculations, and traced the variable values on scratch paper (called doodles) got an average of 75% of the questions correct compared to an average of 50% for those who didn't.

One weakness of this study is that several of the questions were *non-idiomatic*, meaning that the code violated expectations like not looping through all elements when comparing values in two arrays, and only students at one institution were warned that the code could be tricky.

2.1.3 Foundational CS 1 Assessment Instrument (FCS1)

Allison Elliott Tew created the first validated language independent test of CS1 knowledge (Tew & Guzdial, 2010). Her study involved 952 undergraduate students from two different institutions who were in an introductory computer science course. Each student took two exams, one in a pseudo-code and one in the language used in their course: Matlab, Python, or Java. The average score on the pseudo-code exam was less than 34%. The average score on the language specific exam was less than 49%. Students had the most success when answering questions about math operators and conditions and had the most trouble with questions about function parameters, function return values, and recursion.

2.1.4 ITiCSE 2013 Working Group

The ITiCSE 2013 working group revisited the McCracken Working Group Study and also asked students to write code (Utting et al., 2013). The study was conducted at 12 institutions in 10 countries with 418 first year students who had completed at least one university-level introductory programming course. The amount of prior programming experience varied, the range was four to ten European Credit Transfer System (ECTS) credits with a weighted average of seven. One ECTS credit is approximately 25-30 hours of student work, including formal teaching as well as student practice and study.

They measured the students' knowledge of introductory programming concepts using the FCS1 developed by Tew and Guzdial (Tew & Guzdial, 2010). The programing task was to implement four methods in a Clock class: the tick method that added one second to the current time, the compare method that compared one time to another, the

add method that added two times, and the subtract method that subtracted two times. Students were provided the skeleton code for the class and a test harness to check their results. The test harness had 8-10 test cases per method. The students completed, on average, 2.72 methods out of the 4. Students answered an average of 11.35 (42.02%, σ = 4.711) out of 27 questions correctly on FCS1. There was a positive correlation between the students' score on the clock task and their score on FCS1. Four of the six (67%) instructors involved felt that the students' results matched their expectations of their students on that task and exam.

2.1.5 Summary of Multi-Institution Studies of Novice Programmers

Several multi-institution and multi-national studies show that many undergraduate students perform poorly on tests of even introductory programming concepts. The worldwide average success rate in introductory courses is about 67% (Bennedsen & Caspersen, 2007; Watson & Li, 2014). Novice programmers typically have fragile knowledge that is context specific, focused on surface-level features, and use poor general solving strategies (Robins, Rountree, & Rountree, 2003).

2.2 How People Learn

How do people learn anyway? There are at least three very different views about how people learn (Greeno, Collins, & Resnick, 1996). In the *behaviorist* or *empiricist* view learning is about strengthening positive responses with positive reinforcement (rewards) and reducing negative responses with negative reinforcement (punishment or ignoring the behavior). Skinner was a proponent of this approach. In the *cognitive* or

rationalist view the learner must construct his or her own understanding by making sense of the information and by building a mental representation. This view is based on research in cognitive science. Piaget's theory of constructivist learning is an example of this view (Wadsworth, 1989). In the *situative* or *pragmatist* view knowledge is distributed between the people, the objects in the environment, and the community. Dewey is classified as a pragmatist (Dewey, 1959). Lave and Wenger's work on how learning occurs in communities of practice also belongs in this category (Lave & Wenger, 1991). Vygotsky's work emphasized the social and cultural aspects of learning (Berk & Winsler, 1995). While I do believe that learning is situative, and has an important social component, in this work I am mostly building on the cognitivist view of learning.

2.2.1 The Cognitive Theory of Learning

The cognitive theory of learning is based on research in cognitive science. Three of the main principles from this research are shown below (Clark & Mayer, 2011).

- People have dual channels for processing visual information and auditory information. People can handle both visual and auditory information at the same time without overloading either system. For example, people are able to easily integrate the audio and visual information from a movie.
- People can only process a limited number of items (about 5-9) in each channel at one time. Information in working memory only persists for a short while. In order to recall new information for longer periods of time it must be stored into long term memory.
- People must be actively involved for learning to occur. They must attend to the relevant material, organize the material into a coherent structure, and integrate it

into what they already know which is stored in long-term memory. Long-term memory is unlimited and the information in it can be stored indefinitely.

2.2.2 Cognitive Load Theory

Cognitive load theory (CLT) is an instructional theory based on knowledge about human cognitive architecture as described in the cognitive theory of learning. It was developed by John Sweller in the late 1980s (Sweller, 2010). It can be used to improve the design of instructional material. There are three types of cognitive load described in the theory: intrinsic cognitive load, germane cognitive load, and extraneous cognitive load. *Intrinsic* cognitive load is the amount of load due to the difficulty of the problem being solved. *Extraneous* cognitive load is the load added by the complexity of the instructional materials. *Germane* cognitive load is the load devoted to the processing, construction and automation of schemas in long-term memory.

The goal is to design instructional material that frees up working memory to allow learning to occur by reducing the extraneous load and focusing resources on the germane load to allow for the construction of schemas. If the instructional material overloads working memory then learning is impeded. If the intrinsic load is too high then the problem can be broken into smaller and simpler sub problems.

Many studies have found effects that are consistent with cognitive load theory. One effect is known as the *expertise reversal effect*, which means that instructional strategies that are effective with new learners can actually impede experts' learning (Kalyuga, Ayres, Chandler, & Sweller, 2003). Another effect is the *split attention effect*, which means that learning is impeded when the learner must split his or her attention

between instructional materials that use the same channel (Chandler & Sweller, 1992). An example of this is when a student must read a program and a textual description of the program that appears below the program. This is an example of increasing the extraneous cognitive load. The textual description should be integrated into the program using comments or the description could be given in audio, so that the visual channel is not overloaded.

Another effect is the *modality effect*. Since humans can process both visual and audio information at the same time, learning is improved by providing audio narration of complex graphical information rather than a textual description. A review of 21 experiments comparing learning from printed text and graphics versus audio narration and graphics found that students who received audio narration and graphics performed better on problems that were similar to the experimental problems than the students who received text and graphics (Mayer, 2005). The median effect size was large (.97).

2.2.3 Worked Examples

One of the most well-known effects predicted by cognitive load theory is the *worked examples* effect. A *worked example* is a detailed description or demonstration of how to solve a problem. Sweller proposed a "*Borrowing and Reorganizing Principle*" which means that the way that humans build long-term knowledge is by imitating others (Sweller, 2004). Worked examples are an efficient way of doing that. Studies have shown that worked examples improve learning in algebra, physics, and programming. However, students don't always learn from worked examples. They learn best when the

worked examples are interleaved with practice problems that are similar to the worked examples (Trafton & Reiser, 1993).

2.2.3.1 Worked Examples and Algebra Problems

In the first experiment to show the effect of worked examples, Sweller and Cooper had one group of students solve eight algebra problems while another group received four pairs of worked examples and practice problems to solve (Sweller & Cooper, 1985). The group that received the paired worked examples and practice took much less time to complete the training task, took less time to complete the posttest, and did better on the posttest. Several other studies have found similar benefits of worked examples (Cooper & Sweller, 1987; Zhu & Simon, 1987).

2.2.3.2 Worked Examples and Physics Problems

Ward and Sweller (1990) conducted 5 experiments using worked examples in a high school physics classroom. They found that worked examples can be effective in a high school classroom, that the worked examples with interleaved practice problems group did better on transfer tasks than the practice only group, that worked examples that required the learner to split his or her attention between text and graphics didn't result in any improvement in learning, integrating text and graphics in the worked examples did result in an improvement in learning, and that extra textual information that doesn't have to do with the example can actually impede learning (M. Ward & Sweller, 1990). The authors hypothesized that effective worked examples must direct the learner's attention and reduce cognitive load.

2.2.3.3 Worked Examples and Programming Recursive Functions

Pirolli and Anderson reported on observations of two college students and an eight year old as they learned to program recursive functions (Pirolli & Anderson, 1985). The three subjects all used examples when writing their first recursive programs. The three subjects performed best when the examples were similar to the code that they had to write. The subject who performed the best after the training focused on the structure of code in the example, not on the execution process.

2.2.4 Interleaved Examples and Practice

Trafton and Reiser found that it is important for the user to apply the information in the worked example to a similar problem with interleaved worked examples and practice problems in order to improve learning (Trafton & Reiser, 1993). Students who had interleaved examples and similar practice problems solved the target problems more quickly and accurately than those who had blocked examples and practice, which means a block of examples followed by a block of practice. This experiment provided support for the *Knowledge Construction* model of learning, which argues that the learner must apply new knowledge to solve a problem in order to retain the new knowledge. This is in contrast with the *Example Generalization* model of learning which argues that the learner can generalize knowledge from simply studying multiple examples.

2.2.5 Summary of Worked Examples

Worked examples are particularly useful for initial cognitive skill development, such as in learning to program (Atkinson et al., 2000). Worked examples lower cognitive

load and allow the learner to "borrow" knowledge from others and to reorganize it into retrievable knowledge in their own long-term memory. Another argument in favor of worked examples is that students prefer learning by studying examples vs learning by reading text (LeFevre & Dixon, 1986).

2.3 Intelligent Tutoring Systems

A study comparing conventional classroom instruction, with one teacher for about 30 students, versus assigning a good human tutor per every one to three students, showed that the students in the tutor condition scored two standard deviations higher than the students in the conventional classroom (Bloom, 1984). However, it is expensive to provide a good human tutor for every one to three students. One of the goals of artificial intelligence researchers, has been to achieve similar results with software systems known as Intelligent Tutoring Systems (ITS). These software tutors have both an inner loop and outer loop (Vanlehn, 2006). The inner loop executes once per step taken by a student when solving a problem and can provide feedback and hints. This is similar to my intra-problem adaptation, except that my hints are implied rather than explicit. It can also be used to assess the student's mastery of the concepts and update the student model, which is used by the outer loop to select the next problem based on student performance. This adapts the selection of the next problem, which is different from my inter-problem adaptation in which I modify the difficulty of the next problem based on the user's past performance. Tutors typically require a student to master one section before going on to the next section.

In the 1980's Anderson created an Intelligent Tutoring System for programming in LISP. This system used a model-tracing approach in which the student's current solution is compared to solution paths and buggy paths (which represent common errors that students make) (Le, Strickroth, Gross, & Pinkwart, 2013). If the student is on a buggy path then feedback and hints can be provided to guide the student back to a solution path. Students using the LISP tutor, which integrated the tutor into the programming environment, took 30% less time and scored a standard deviation higher than students who used a standard LISP development environment to solve the same problems (John R. Anderson et al., 1995). While several ITS have achieved positive results, they have not been widely used (John R. Anderson et al., 1995). One problem is the development time. It can take 23 hours to create one hour of instruction for a simple ITS, and most require 100 hours or more (Corbett et al., 1997; Folsom-Kovarik et al., 2010). Other problems include a misalignment between the ITS curriculum and what educators actually teach and the difficulty of customizing the material (John R. Anderson et al., 1995). In contrast, Parsons problems can be authored in minutes, instructors can assign problems as desired, and new problems can be added at any time.

2.4 Data-Driven Hint Generation

Due to lengthy development time for an Intelligent Tutoring System (ITS), several researchers have tried to generate hints from past student data. This approach has been used to generate hints for a logic proof tutor (Barnes & Stamper, 2008). One group generated a Markov Decision Processes (MDP) that represents all the encountered approaches to a particular problem, and then use the MDPs to generate hints. They were

able to provide highly contextualized hints for about 70% of the moves, using just one semester of data with about 200 students. This technique has also been used to generate meaningful hints for 66% of the incorrect submissions in a programming tutor (Jin et al., 2012) .

Another approach uses state abstraction, path construction, and state reification to generate personalized hints for code-writing problems, even on states that have not been seen before (Rivers & Koedinger, 2017). However, this approach has not yet been tested with students.

While these approaches are promising, they require past student data on the same problem. In contrast, Parsons problems do not require any past student data.

2.5 Active, Constructive, and Interactive Learning Activities

Michelene Chi defined a framework of active, constructive, and interactive learning activities (Michelene T. H. Chi, 2009). In her framework, active means some physical motion such as focusing on the text that you are reading, selecting, or clicking. Constructive is active, but the difference is that some output must be created such as notes or predictions and that output must contain more than just the content of the learning materials. Interactive is some type of give and take with another entity like another person or a computer system that is giving feedback or hints. Her hypothesis is that active learning activities are better than passive, constructive learning activities are better than active, and interactive learning activities are better than constructive. She points out that this may sound like it contradicts findings in using worked examples. However, looking at worked examples can be constructive when students explain the

example to themselves as they study them. This is called *self-explanation*. Chi found that good students produce self-explanations when studying worked examples (M. T. H. Chi, Bassok, Lewis, Reimann, & Glaser, 1989), while poor students do not. Worked examples often convey more information than trying to solve the problem on your own. Cognitive load theory can be used to improve instruction using active, constructive, or interactive activity.

2.6 The Importance of Practice

Practice is essential for learning. It helps the learner focus on, organize, integrate, and retrieve new knowledge from long-term memory. Several studies show the importance of practice in developing expertise (Slobada, Davidson, Howe, & Moore, 1996; Tuffiash, Roring, & Ericsson, 2007). But, it needs to be the right kind of practice. It is possible to spend many hours practicing without any improvement in ability. I sang in a church choir for many years, without much improvement since we sang the same songs and I didn't get feedback to improve my performance.

2.6.1 Zone of Proximal Development (ZPD)

One of the things that drove Vygotsky's work was his desire to help children with physical and psychological problems. Unlike many other psychologists of his time, he believed in the importance of applying his theories. One of his most well-known theories, *the zone of proximal development*, predicts that learning and cognitive development occurs when the learner is given a task that is just beyond what he or she could handle without support. The zone of proximal development is defined as *"the*

distance between the actual development level as determined by independent problem solving and the level of potential development as determined through problem solving under adult guidance or in collaboration with more capable peers (Vygotsky, 1978)" The goal of education should be to keep the learner in his or her zone of proximal development in order to maximize learning.

2.6.2 Deliberate Practice

To improve performance it needs to be *deliberate practice* which means that it focuses on areas where the learner is weaker and includes feedback, which can be used to improve results (K. Anders Ericsson, 2006). For everyday activities, such as learning to drive or learning to touch-type, 50 hours of practice is usually enough to reach acceptable performance levels. The *power law of practice* shows that continued practice has a diminishing effect (Rosenbaum, Carlson, & Gimore, 2001). While practice leads to large improvements during initial learning, the benefit decreases as proficiency increases. It can take years or decades to reach expert levels of performance for some activities. Chess experts accumulate over 50,000 "chunks" or "patterns" over years or decades to allow them to determine the appropriate moves based on the current position of the pieces in a chess game (H. A. Simon & Chase, 1973). It can take ten years to become an expert programmer (Winslow, 1996)

Deliberate practice requires repetitive practice tasks that are just beyond the learners' current ability and also requires timely feedback that can be used to improve performance. A "coach" usually selects the practice tasks and provides the feedback. In a study of expert violinists, all of them reported the same amount of practice time each

week (over 50 hours), but the very best violinists spent more of that practice time on *deliberate practice* (K. A. Ericsson, Krampe, & Tesch-Romer, 1993). A study of elite soccer players found that the elite players didn't spend any more time in practice or performance than non-elite players, but spent more of that time on *deliberate practice* (P. Ward, Hodges, & Starkes, 2004).

2.6.3 Desirable Difficulties

New information is not just stored or copied into long-term memory; it is related to and integrated into what learners already know. Retrieval of information depends heavily on the context, which can limit our ability to transfer information from one context to another. Retrieving information from long-term memory increases our ability to recall it in the future. Long-term learning is improved by techniques that help us store and retrieve information in multiple contexts. *Desirable difficulties* are those that help learners store and recall information in multiple contexts (E. L. Bjork & Bjork, 2011). One key idea of this work is that improving the learner's performance while learning can actually decrease long-term learning, and conversely techniques that reduce the learner's performance while learning can actually lead to long-term retention and better recall. Some techniques that result in desirable difficulties are spaced practice over time rather than massed practice (Druckman & Bjork, 1991; Rohrer & Taylor, 2006). Other types of desirable difficulties include interleaving different subjects, varying the environment during practice, and using tests for learning rather than just for assessment.

2.6.4 Adaptive Practice

Corbalan, Kester, and van Merrienboer found that dynamically adaptive practice, where the practice problems are adapted based on the learners prior performance, improves learning, takes less time, and increases engagement (Corbalan, Kester, & Merrieonbeor, 2008). This is not surprising since adaptive practice is more likely to keep the learner in Vygotsky's zone of proximal development (Berk & Winsler, 1995).

Soloway, Guzdial, and Hay called for a change in Human Computer Interaction (HCI) from User-Centered Design to Learner-Centered Design (Soloway, Guzdial, & Hay, 1994). In particular they called for more scaffolding which supports learners as they try to accomplish a new task (Rogoff, 1990). They describe several types of scaffolding including limiting the starting task to not be overwhelming, modeling behavior, providing hints, encouraging reflection, and encouraging meta-cognition. To be most effective, scaffolding should fade as the learner develops expertise. In order words, the system should adapt to the learner's performance. However, as the authors say, *"Build learner-centered software! Easy to say, hard to do."*

Intelligent Tutoring Systems (ITS) provide scaffolding to help the learner solve the current problem and also select the next problem based on a model of the student's mastery of the desired concepts, however they take a great deal of time to build (Corbett et al., 1997) and are not widely used (John R. Anderson et al., 1995). Adaptive Parsons problems use intra-problem adaptation to provide implicit hints to help the learner solve the current problem and inter-problem adaptation to modify the difficulty of the next

problem. They are easy to create and are available in interactive ebooks that hundreds of institutions already use.

2.7 Reducing Cognitive Load in Computer Science

Computer science education often depends on whole task learning where the student is required to program a complete solution to a problem. Whole task learning is appealing in that it is seen as authentic (Shaffer & Resnick, 1999) and can help students transfer knowledge and skill to real life contexts. However, whole task learning can have a high cognitive load, which can quickly overwhelm novices and actually impede learning and decrease motivation. *Scaffolding* can help lower the cognitive load for novice learners and allow learners to solve a problem that they would not be able to solve without support (Hmelo & Guzdial, 1996). Scaffolding supports performance and fades as the learner develops knowledge and skill. An example of scaffolding is helping a child who is learning to read sound out an unfamiliar word. Other examples of scaffolding are hints, checklists, and feedback. Worked examples are also a type of scaffolding, which lower the cognitive load of the task since the learner can "borrow" knowledge from the example. It is important for scaffolding to fade as the leaner develops skill. Supports that beginners need to reduce the cognitive load of a task by providing external guidance in the form of scaffolding can actually increase the cognitive load for experts, because it is redundant with their internal schema-based knowledge. This is known as the *expertise reversal effect*.

2.7.1 Multiple-Choice Questions

Multiple-choice questions are one way to provide practice problems that test a learner's understanding. One drawback to multiple-choice questions is that the learner can guess the answer, though one study found evidence that only about 10% of the students showed evidence of guessing based on the transcripts from their observations (Lister et al., 2004).

2.7.2 Parsons Problems

Parsons problems are a type of code construction problem that require the solver to put mixed up code blocks containing one or more lines of code into the correct order to create a correct program. Several researchers have investigated Parsons problems and they have used a variety of names to refer to them: Part-complete solution method (PCSM) (Garner, 2007), Code Mangler Problems ((Cheng & Harrington, 2017), Parson's Programming Puzzles (Parsons & Haden, 2006), and Parsons puzzles/problems (Helminen, Ihantola, Karavirta, & Malmi, 2012; Ihantola & Karavirta, 2011). Dale Parsons originally called them Parson's Programming Puzzles, but she says that "Parson's" was a mistake since her last name is Parsons and she prefers Parsons Programming Puzzles. I have used Parsons problems due to the length of "Parsons Programming Puzzles" and the fact that several more recent researchers have used that term.

Several variations of Parsons problems have been studied:

- Some provide just the correct code blocks that are mixed up and have to be placed into the correct order (Harms, Rowlett, & Kelleher, 2015).

- Some provide all the correct code blocks as well as unneeded blocks that contain code with syntax, sematic, or logic errors. These extra blocks are called *distractors* (Parsons & Haden, 2006) (Denny et al., 2008). There are two sub-types of Parsons problems with distractors. In the *paired type* the correct code block and incorrect code block are shown as pairs so that the solver only has to choose between them. In the *un-paired type* the code and incorrect code blocks are not shown in pairs, but are all jumbled together (Denny et al., 2008).
- Some require the solver to *indent* the code blocks as well as order them. These are called *two-dimensional Parsons problems* (Helminen et al., 2012; Ihantola & Karavirta, 2011; Karavirta, Helminen, & Ihantola, 2012).
- Some provide some or most of the correct code that is needed, but the solver must provide (type or write by hand) some of the needed code or at least add symbols to indicate the block structure such as adding curly braces in Java code to indicate a block of code (Denny et al., 2008; Garner, 2007). This is like requiring the user to indent code in Python to indicate a block.
- Some provide part of the solution code already in order, and the solver only needs to add code blocks to complete the solution (Garner, 2007).

Dale Parsons and Patricia Haden suggested that Parsons problems would be engaging problems and would help novices practice and master syntax and basic

programming constructs in Turbo Pascal (Parsons & Haden, 2006). They included distractor code with syntactic and semantic errors in their problems. Most undergraduate students (82%) in their small study (n=17) reported that the puzzles were useful or very useful for learning Pascal on a post survey. However, most students also wanted better feedback when they made an error and the ability to compare their solution with the correct solution. They used a web-based tool called Hot Potatoes to create the Parsons problems as shown in Figure 4.

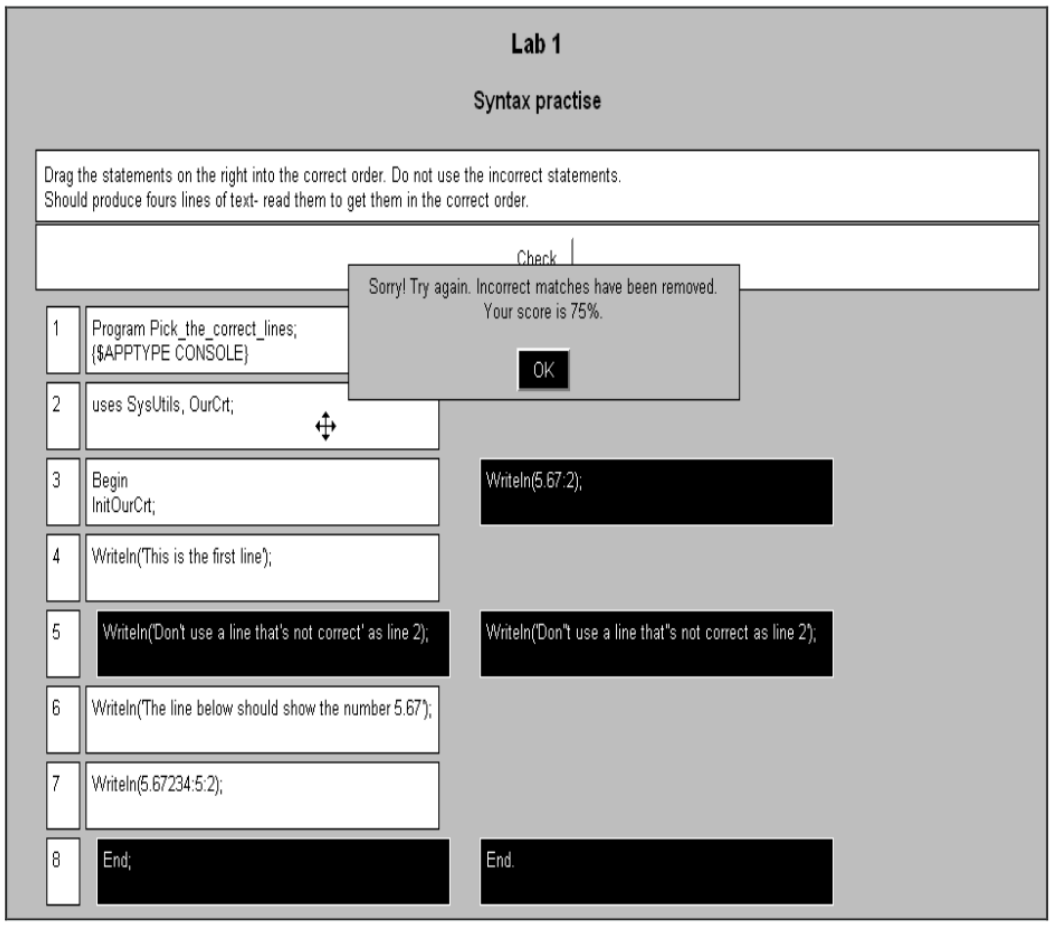


Figure 4. A Parsons problem in Hot Potatoes. © 2006 Australian Computer Society, Inc. Used with permission.

Garner created and studied the use of a tool, CORT, that allowed the solver to add correct and/or distractor lines to a partially complete Visual Basic program and then execute the solution (Garner, 2007). He created 17 Parsons problems using CORT. Figure 5 shows an example of a Parsons problem in CORT. He observed eight students as they solved some of the Parsons problems using the tool. He found evidence that the problems with only the correct code and no distractors were easiest and the problems that

provided some of the correct code, but also required the solver to write some code were the hardest.

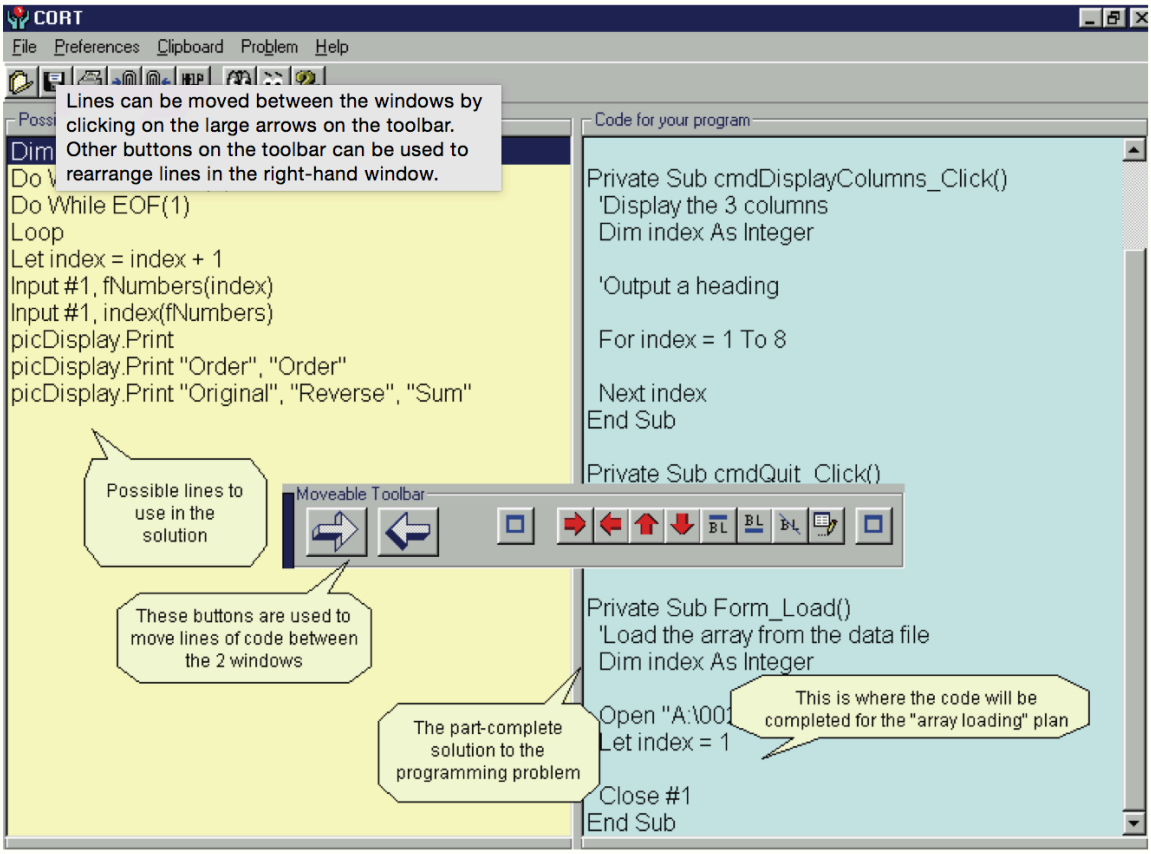
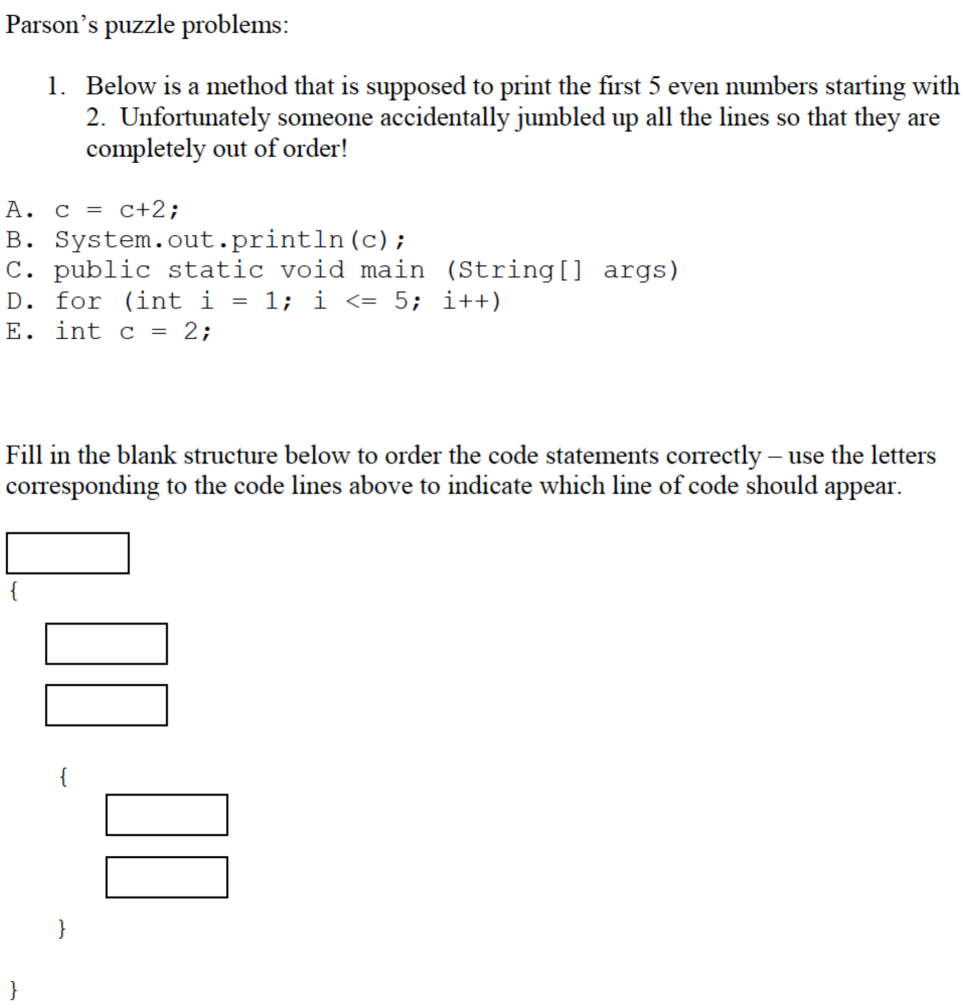


Figure 5. A Parsons problem in CORT. © 2007 Informing Science Institute. Used with permission.

Denny, Luxton-Reilly, and Simon explored Parsons Problem as a possible replacement for requiring students to write code on paper exams (Denny et al., 2008). They argued that Parsons problems would be quicker to grade and result in more consistent grades between markers. They created 5 variants of Parsons problems 1) one with just the correct code and the block structure provided 2) one with just the correct code and no block structure provided 3) paired distractors and correct code with structure

provided 4) paired distractors and correct code with no block structure provided 5) un-paired distractors and correct code with no block structure provided. Figures 6-9 show the first 4 variants.



Parson's puzzle problems:

1. Below is a method that is supposed to print the first 5 even numbers starting with 2. Unfortunately someone accidentally jumbled up all the lines so that they are completely out of order!

```
A. c = c+2;
B. System.out.println(c);
C. public static void main (String[] args)
D. for (int i = 1; i <= 5; i++)
E. int c = 2;
```

Fill in the blank structure below to order the code statements correctly – use the letters corresponding to the code lines above to indicate which line of code should appear.

```
[          ]
{
    [          ]
    [          ]

    {
        [          ]
        [          ]
    }

}
```

Figure 6. Parsons problem variant #1 studied by Denny et al. Used with permission.

2. Below is a method that is supposed to print the first 5 even numbers starting with 2. Unfortunately someone accidentally jumbled up all the lines so that they are completely out of order! (Letters are shown in front of each line to allow you to refer to them)

```
A. c = c+2;
B. System.out.println(c);
C. public static void main (String[] args)
D. for (int i = 1; i <= 5; i++)
E. int c = 2;
```

Below, write the method correctly to print the first 5 even numbers starting with 2. Rather than writing out all the lines of the code, you can just write down the letter of the line (as indicated above). You will find that you need to add all your own curly braces, that is { and } to make correct and legal Java code.

Figure 7. Parsons problem variant #2 studied by Denny et al. Used with permission.

3. Below are some lines of Java code that you will use in solving this problem. Before each line is a letter – which just exists to allow you to refer to that line (rather that writing it out again for yourself).

```
A. c = c+2;
B. c = c+1;
C. System.out.println(c);
D. System.out.println(c+2);
E. public static void main (String[] args)
F. private static boolean even (String[] args)
G. for (int i = 1; i <= 5; i++)
H. for (int i = 0; i <= 5; i++)
I. int c = 2;
J. int c = 1;
```

Fill in the blank structure below to order SOME of the above code statements correctly – to print the first 5 even numbers starting with 2. Use the letters corresponding to the code lines above to indicate which line of code should appear.

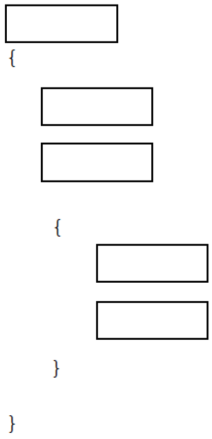


Figure 8. Parsons problem variant #3 studied by Denny et al. Used with permission.

4. Below are some lines of Java code that you will use in solving this problem. Before each line is a letter – which just exists to allow you to refer to that line (rather that writing it out again for yourself).

```
A. c = c+2;
B. c = c+1;
C. System.out.println(c);
D. System.out.println(c+2);
E. public static void main (String[] args)
F. private static boolean even (String[] args)
G. for (int i = 1; i <= 5; i++)
H. for (int i = 0; i <= 5; i++)
I. int c = 2;
J. int c = 1;
```

Below, write the method to print the first 5 even numbers starting with 2. Rather than writing out all the lines of the code, you can just write down the letter of the line (as indicated above). You will find that you need to add all your own curly braces, that is { and } to make correct and legal Java code. You may not need to use all of the lines of code provided – just select the ones that you need. DO NOT WRITE YOUR OWN CODE – SELECT FROM THE PROVIDED LINES ABOVE!

Figure 9. Parsons problem variant #4 studied by Denny et al. Used with permission.

Six students solved the five variants during a think aloud observational study. The students had to write the labels of the code in the correct order. The think aloud observations revealed that Parsons problems with un-paired distractors for nearly every line of code were too difficult for the students (variant 5). Parsons problems with paired distractors (variants 3 and 4) were easier. Providing the structure of the code, the number of statements in a block indicated by curly braces and the indention (variant 2), also made Parsons problems easier to solve. They also found that having the students just write the labels down rather than write the code made it harder for the students to do the task. As one student said, "*1/2 way through and you can't remember what the letters for the top half stand for*." This quote shows why our approach of having students drag the blocks

into order to form a solution is easier for them than having them write the block labels in order.

They had 74 undergraduate students solve a Parsons problem, a code-writing problem and a code-tracing problem on an exam. The Parsons problem paired the correct and distractor code by adding vertical space below each pair as shown in Figure 10. The students had to write the code in the correct order and add the curly braces to indicate the block structure in order to create a Java method that would remove all occurrences of the character 'a' from the passed string (word).

```
return result;
return word;

String result = "";
String result;

if (word.charAt(i) == 'a')
if (word.charAt(i) != 'a')

for (int i = 0; i < word.length(); i++)
for (int i = 0; i < word.length; i++)

result = result + word.charAt(i);
result = word.charAt(i);

private String removeAllAs(String word)
private String removeAllAs(word)
```

Figure 10. Parsons problem with paired distractor and correct code to remove all 'a's from a string. © 2008 Association for Computing Machinery, Inc. Reprinted by permission.

They found a notable correlation (Spearman's r^2 of .53 which explains 73% of the variance), between the score on the code-writing and Parson problem. The lowest quartile of students did the worst on code tracing, better on code writing, and better still on

solving Parsons problems. Some students who were able to solve the Parsons problem still didn't understand how the code worked. They were able to figure out the correct order from general knowledge about the structure of a function like the declaration goes at the top and then variables are declared and a return statement is the last statement in the function.

Morrison et al. provided further evidence that Parsons problems are a more sensitive measure of learning, i.e., that a Parsons problem might detect a learning difference between students that might not appear in a code-writing activity (Morrison, Margulieux, Ericson, & Guzdial, 2016).

Ihantola and Karavirta developed open source software called js-parsons to allow learners to solve two-dimensional Parsons problems and then observed four experts using the tool to solve 10 Parsons problems (Ihantola & Karavirta, 2011). In two-dimensional Parsons problems, the code has to be both ordered correctly and indented correctly. When the user asks for feedback, the js-parsons software displays blocks that are in the correct order with the correct indentation with a green background, blocks in the incorrect order with a red background, and blocks that are in the correct order but have the wrong indentation with a red background with an additional red mark on the left side as shown in Figure 11.

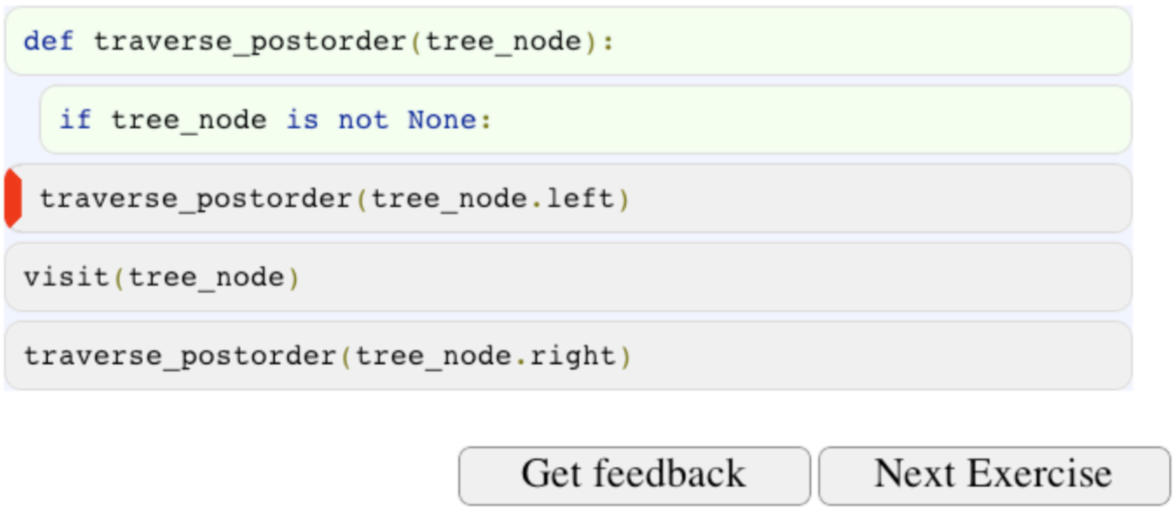


Figure 11. Feedback in js-parsons. The blocks with a green background are in the correct order with the correct indentation. The block with the red background with the left edge highlighted is in the correct order, but needs to be indented. The last two blocks with a red background need to be swapped. © 2011 Informing Science Institute. Used with permission.

The authors found that two-dimensional Parsons problems are more difficult than the original Parsons problems where the code blocks only had to be dragged into the correct order and not indented. The authors suggested that Parsons problems could be used to help novices learn to recognize common algorithms like finding the smallest value in an array. They found evidence of the *expertise reversal effect* from educational psychology (Kalyuga et al., 2003). One expert said, "*This is more difficult than writing code when the expected solution does not match one's own mental model of the algorithm*". They also found that experts didn't always solve the Parsons problems from top to bottom, but instead assembled major control structures first like function declarations and loops. This highlights an advantage to a tool like js-parsons versus a tool like Hot Potatoes, since it is easy to drag a new block between two already placed

blocks in js-parsons, while this is impossible in Hot Potatoes.

Helminen, Ihantola, Karavirta, and Malmi also examined how undergraduate students solved two-dimensional Parsons problems as well as the type of problems they encountered (Helminen et al., 2012). They found that some students repeated the same incorrect solutions.

Karavirta, Helminen, and Ihantola created MobileParsons to allow Parsons problems to be solved on mobile devices (Karavirta et al., 2012). They also provided additional feedback, but did not test the additional feedback. They added new features (Ihantola, Helminen, & Karavirta, 2013), which allow more than one possible solution. They also allow the learner to toggle or tap through several possible values for part of a statement, like the comparison operator in a conditional. The learner's solution is evaluated by running unit tests provide by the author. The following types are supported in the areas that can be toggled: variable names, Booelan values (True or False), math operators (+), comparison operators (>=), logical operators (and, or) and numeric ranges. Figure 12 shows the areas that can be toggled outlined in red on the left and a partial answer on the right.

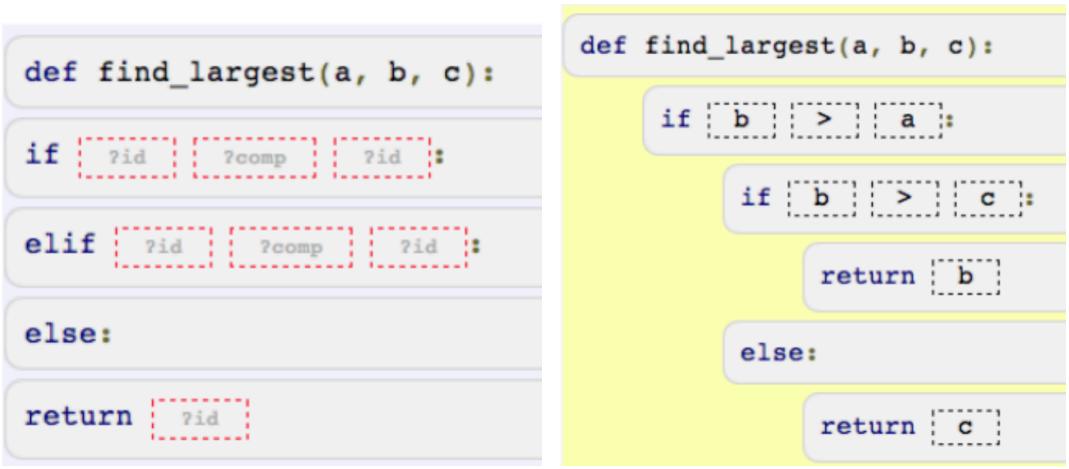


Figure 12. Initial state (left) and a partial solution (right). © 2013 Ihantola, Helminen, and Karavirta. Used with permission.

This group also conducted a study of over 400 undergraduate students in an introductory programming course where one group received the usual feedback from js-parsons (red background to indicate blocks that are out of order and red left border highlights to indicate an error in indention) and the other received feedback from running the code against unit tests (Helminen, Ihantola, Karavirta, & Alaoutinen, 2013). The Parsons problems did not contain any distractors. Some problems were solved on their first attempt (either asking for feedback or running the code). On problems that the students didn't solve on their first attempt, the group who received execution-based feedback (feedback from executing the code) took longer to solve the problem and requested feedback less often than the group that received the typical feedback from js-parsons (red highlights to indicate out of order code and incorrect indention). This is likely due to the difficulty novice students have with understanding compiler errors and debugging. Some students took many more attempts to solve a Parsons problem than the authors expected (a maximum of 409 attempts). On a survey sent to all the students, the

lower ranked students found the Parsons problems more difficult than the higher ranked students. The most experienced students felt that the Parsons problems were too easy and useless, which again shows the *expertise reversal effect (Kalyuga, 2007)*.

Harms, Rowlett, and Kelleher compared learning from solving Parsons problems with only correct code to following step-by-step tutorials with a video demonstrating each step (Harms et al., 2015). Both the Parsons problems and tutorials were integrated into Looking Glass, which is a drag and drop programming environment that is a descendant of Storytelling Alice (Kelleher, Pausch, & Kiesler, 2007). They measured learning, enjoyment, and transfer. They found that the Parsons problem solvers complete the learning task more quickly (23% less time) than tutorial takers and also did 26% better on transfer tasks. In their formative study, they observed a marked preference for Parsons problems versus tutorials, but the results on the attitudinal survey didn't show any statistical difference in the between-subjects study. They suggest using a within-subjects study to test this further. The Parsons problem solvers also reported higher mental effort to complete the task than the tutorial followers, which is consistent with *desirable difficulties* leading to increased learning (E. L. Bjork & Bjork, 2011). Figure 13 shows a Parsons problem in Looking Glass.

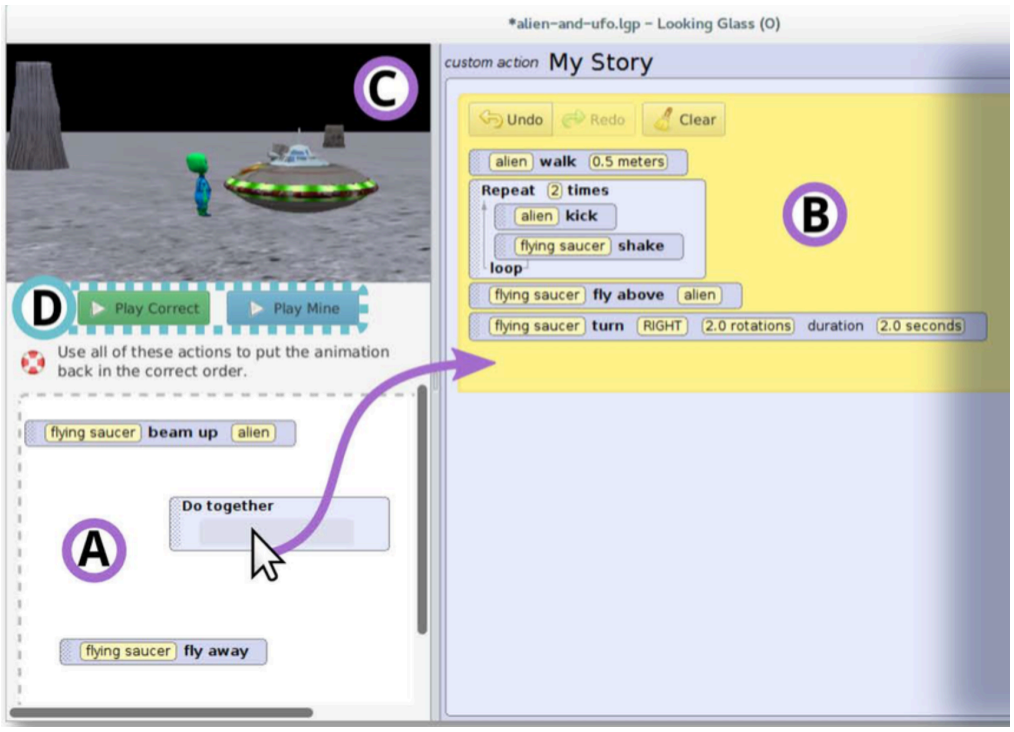


Figure 13. A Parsons problem in Looking Glass. © 2015 IEEE.

2.7.3 Summary of Research on Parsons Problems

Over 80% of students found Parsons problems useful during initial learning of programming concepts (Parsons & Haden, 2006). Researchers have explored a variety of Parsons problems and have provided evidence that Parsons problems with distractors are harder than those without distractors (Garner, 2007; Harms, Chen, & Kelleher, 2016) and problems with more distractors are harder than those with less (Denny et al., 2008). Parsons problems that require indention are harder than those that do not (Ihantola & Karavirta, 2011). Parsons problems with visually paired distractors and correct code are easier than un-paired distractors, which are randomly mixed in the with the correct code blocks (Denny et al., 2008). Students also enjoy solving Parsons problems more than taking a tutorial (Harns, 2015).

2.8 Summary of Prior Research

Learning to program can be difficult. Introductory programming courses often require learners to write lots of programs. Writing programs can overwhelm working memory and impede learning. Parsons problems with distractor code containing syntactic and semantic errors are a type of code completion activity that should have a lower cognitive load than fixing code with syntactic and semantic errors or than writing the equivalent code. Researchers have found that over 50% of the errors in novice programmer's solutions were program composition errors (Spohrer & Soloway, 1986). They suggested that educators teach novices strategies for how to put the pieces of program code together (Soloway, 1986). Parsons problems should help novices learn strategies for putting program code together. Parsons problems with distractors that contain syntactic and semantic errors should help students learn to recognize those types of errors.

Intra-problem and inter-problem adaptable Parsons problems should increase completion rates, increase enjoyment, and increase learning compared to non-adaptive Parsons problems due to keeping the learner in his or her *zone of proximal development* in which problems are just beyond the learners' current capability but solvable with support (Berk & Winsler, 1995). Adjusting the difficulty of the current problem (intra-problem adaptation) to match the learner's performance will also lead to *deliberate practice*, which is practice on the things that are just outside the learners current ability and has been shown to improve performance (K. Anders Ericsson, 2006). Making the next problem harder (inter-problem adaptation) if the learner has solved the current

problem too quickly should provide *desirable difficulty*, which improves long-term learning (E. L. Bjork & Bjork, 2011).

CHAPTER 3. INITIAL INVESTIGATIONS OF PARSONS PROBLEMS

The CSLearning4U research group at the Georgia Institute of Technology started studying ebooks by adding material and interactive features to the *How to Think Like a Computer Scientist – Interactive Edition* ebook (B. Miller & Ranum, 2013; B. N. Miller & Ranum, 2012). Many colleges and universities, including Luther College, Duke University, the University of Illinois, and Heidelberg University use this ebook (at https://runestone.academy/runestone/static/thinkcspy/index.html). The web site states that over 850,000 people have used this ebook (B. Miller & Ranum, 2013). The ebook originally included the following features:

- Videos
- Executable and modifiable Python code (called Active Code)
- A code visualizer and stepper (called Code Lens)
- Multiple-choice questions with the same feedback regardless of the answer

Our group added the following new features:

- Audio tours of code
- Parsons problems
- Multiple-choice questions with feedback specific to the selected answer

The audio tours stepped through the code and explained highlighted lines of code using audio, thus leveraging the multi-modal principle. The Parsons problems were two-

dimensional Parsons problems implemented using js-parsons (Ihantola & Karavirta, 2011). The multiple-choice questions gave feedback specific to the selected answer, whereas originally the feedback was the same regardless of the selected answer.

I added 11 Parsons problems to a chapter of the *How to Think Like a Computer Scientist – Interactive Edition* ebook. An example is shown in Figure 14. These Parsons problems all used turtle graphics, in which a simulated robotic turtle draws as it moves and understands simple commands like *forward(amount)* and *left(degrees)*. Seymour Papert envisioned the simulated turtle to be a type of "object to think with" (Papert, 1980).

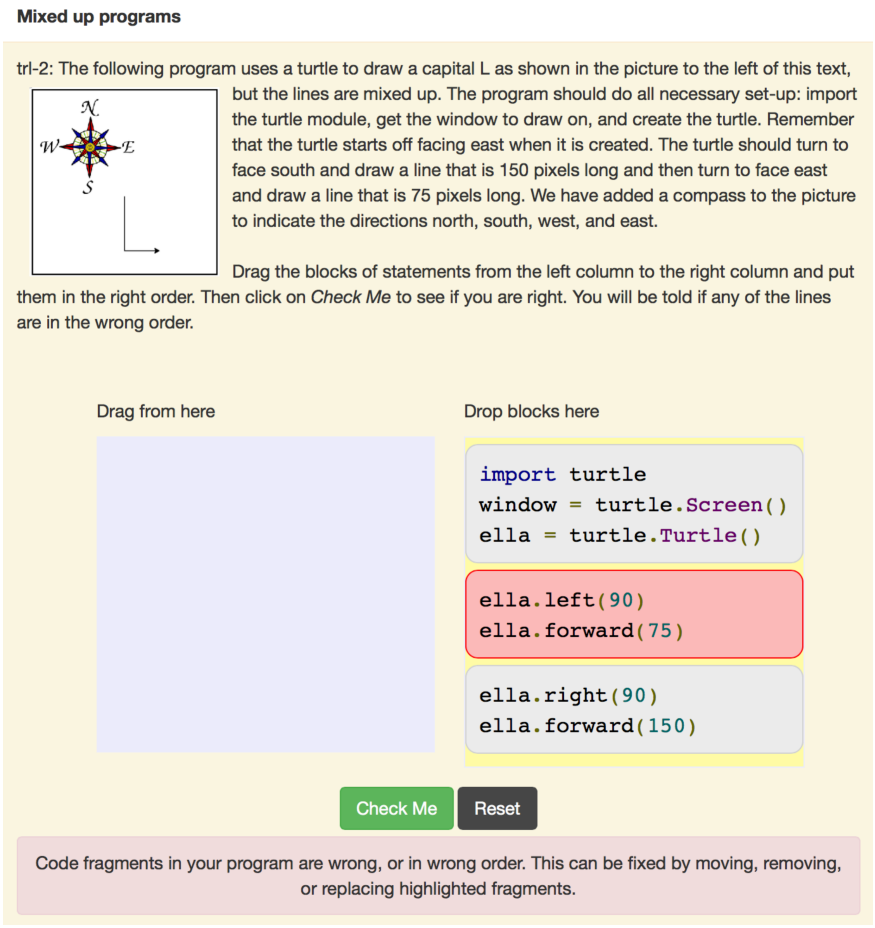


Figure 14. A Turtle Graphics Parsons problem

The Parsons problems included only correct code and each problem had three to eight code blocks as shown in Table 2 below. Problems 8 through 11 were all two-dimensional Parsons problems, which means that the learner had to indent the blocks as well as place them in the correct order.

Table 2. Problem number and number of blocks in the 11 Parsons problems

Problem	# Blocks	What it Draws
1	3	A capital L
2	5	A checkmark
3	5	A line to the west
4	5	A capital L in white on a blue background
5	5	A capital T in white on a green background
6	5	A capital L in blue with one turtle and a line in orange to the west with another
7	6	A line in blue to the north with one turtle and a line to the east with another
8	6	An equilateral triangle
9	6	A rectangle
10	7	A circle of 10 turtles facing out using the stamp function
11	8	Three turtle shapes stamped in a line

The rest of this chapter is adapted from a paper presented at the International Computing Education Research (ICER) conference (B. J. Ericson, Guzdial, & Morrison, 2015).

3.1 Research Questions

Our research questions were 1) *would readers use the new features*, 2) *would some features be used more than others*, and 3) *were the Parsons problems at the right level of difficulty*?

The studies in this chapter were part of my initial exploration of Parsons problems, which is why I didn't include these research questions in my list of research questions for my dissertation. I wanted to verify that teachers and students would try to

solve Parsons problems and find them valuable, before I decided to focus on them for my dissertation.

We expected readers to be familiar with multiple-choice questions and solve those within a few tries. We wanted to see if readers would attempt the Parsons problems, how many attempts it would take them to get them correct, the number of readers who would give up and never solve each Parsons problem, and the number of attempts that readers would make before giving up on solving a Parson problem. We wanted the practice problems to be difficult enough to cause errors which improves learning (E. L. Bjork & Bjork, 2011), but not so difficult that the readers would give up.

3.2 Goals for the Studies

I conducted two studies to examine the use of the added features within the ebook. The first was an observational study of four *teachers* working through one chapter of the ebook after finishing the earlier chapters on their own. The goal of this study was to observe teacher use of the features as they worked through the chapter to determine what difficulties they encountered. This study also allowed me to ask the teachers questions about why they didn't use certain features and what they thought of the features.

The second study was a log file analysis of use of the ebook by *undergraduate and high school students* to determine how many features were actually used, how many students attempted each practice problem, and how many solved each practice problem. The ebook is freely available (B. Miller & Ranum, 2013) and is used by several

universities and some high schools. The log file analysis was informed by the findings from the observational study.

3.3 Observational Study of Teachers

We recruited teachers with less than six months of textual programming experience (in languages like Python or Java) to participate in an observational study. We had 18 teachers fill out the registration and sign the consent forms. We disqualified two teachers due to too much *textual programming experience* (we allowed longer drag-and-drop programming experience).

We asked the teachers to complete the first three chapters of the ebook on their own and then notify us before starting the fourth chapter on turtle graphics and loops. We did the remote observations using the webinar software BlackBoard Collaborate (http://www.blackboard.com/) as the teachers worked through the chapter. We took notes on our observations, but also recorded the sessions so that we could check our notes against the recordings. We instructed the teachers to talk aloud as they worked though the interactive elements and asked what they were thinking if they were silent for more than a few seconds. After they completed the chapter we also asked questions based on our observations. We observed four teachers, since only four teachers notified us that they had completed the first three chapters during the study period. All of the teachers were currently teaching a programming class.

3.4 Findings from the Observations

All four teachers interacted with *all* of the worked examples (Active Code and Code Lens) and practice problems (multiple-choice and Parsons problems) in chapter four. They also all edited the code when directed to by the ebook. They correctly answered most problems correctly in one try, but took two attempts to correctly answer some problems. While these teachers didn't have more than six months of textual programming experience, some of them had taken a college course in programming and several had been teaching programming using drag-and-drop languages such as Scratch or Alice for several years.

Only one of the teachers watched all of the videos. Another teacher said that he had watched the videos in the other chapters, but realized that they covered much the same material as in the text, and said he thought that he could get through the practice problems without watching them. He said, *"The teacher in me wants to watch them (the videos), but the student in me says see how you do without them."*

Only two of the four teachers listened to any of the audio tours. When we asked the other teachers why they hadn't listened to the audio tours, they told us that they hadn't noticed them. After one teacher noticed the audio tours he said, *"The audio tour was interesting – I think that would be helpful if I wanted things to be explained to me."* At the time of this study the *Start Audio Tour* button was gray and was next to the green *Run* button and above two other gray buttons that let the user save and load modified code. The text did not mention the audio tours.

The teachers solved all of the Parsons problems in 1-2 tries. The most common difficulty we observed was that the teachers did not immediately realize that they had to indent statements in the body of a loop, even though the ebook text explained that Python requires the body of the loop to be indented. The Active Code examples also showed that the body of the loop was indented. The text before the Parson problems said that the user would be told if any of the blocks are in the wrong order *or are incorrectly indented*, but the teachers did not notice that text. One teacher even said, *"These need to be indented, but I don't think we can indent here."* After he received an error, he realized he could actually indent the code blocks. Some teachers also got the order of some of the blocks wrong at first, but were able to fix the order in their second attempt. Some teachers read the textual instructions out loud and went back and forth between dragging blocks and reading the instructions. One teacher said, *"I love this. I am curious to see how this works in a classroom."* One teacher said, *"They (the Parsons problems) weren't hard but they gave me a good idea of the flow (import first, name turtle and properties)."*

One teacher dragged code blocks from the left to the right and then dragged some code blocks back to the left before rereading the textual instructions. He used the right side to hold the code segments that he was sure of and the left side to hold the blocks that he still needed to move. This behavior was interesting as we were considering saving space by using just one text area to hold the mixed-up code and having the user reorder the blocks within that area rather than have the user drag blocks from left to right. However, it appears that having the two text areas can reduce the cognitive load on the user when solving the Parsons problem. It allows the user to keep track of what has been

moved and what still needs to be moved. This is an example of *distributed cognition* in which some of the cognitive load is offloaded to items in the environment (Hutchins, 1995).

3.5 Log File Analysis of Ebook Features

In our earlier pilot studies and observational studies we noticed differences between how students and teachers used the ebook (Alvarado et al., 2012). Teachers worked through every practice problem in a chapter, while students skipped some practice problems. Teachers also worked on problems until they got them correct, while some students quickly gave up if they were having problems. Teachers may have completed more problems and continued until they got the problems correct, because they knew they were being observed.

We wanted to investigate how students used the interactive features in the ebook and especially compare the use of the Parsons problems to the other interactive features. Brad Miller, who runs the server, sent us an anonymized version of the log file data in the fall of 2014. This file was over 700 MB and contained data for all student use of the *How to Think Like a Computer Scientist – Interactive Edition* eBook from May 17, 2012 to May 26, 2014. Identifying information for both user and course names were replaced with user and course numbers.

3.5.1 Selecting Courses to Analyze

Instructors can create their own course using the ebook. This gives the instructor access to student progress and activity. While many instructors create their own course, others just use the open access version of the book, which anyone can use.

There were 90 courses represented in the log file data. We selected the 4 courses that had the highest number of entries on which to conduct further analysis. Table 3 shows the course information and number of log file entries for each of the analyzed courses. In addition to the entries related to these 4 specific courses, we also analyzed the data from the open access version of the book between January 1, 2014 to June 1, 2014. The open access version of the book had over 2,000 unique users during this time period. We chose this time period to correspond to the time period of most of the courses and to examine the book after we had added practice features such as the Parsons problems.

Table 3. Courses that were analyzed

Course Number	Start Date	Type	Number of Entries
347	8/25/2013	College	250,909
402	1/26/2014	Unknown	243,362
261	1/6/2014	High School	193,668
137	1/31/2014	College	132,075

3.5.2 Log File Analysis

For each selected course, we calculated the number of unique users who attempted and eventually correctly solved every multiple-choice problem and Parsons problem. We also counted the number of unique users that ran and edited each Active

Code example, clicked the forward button on a Code Lens, played a video, and played an audio tour. We generated the same data for the open access version of the book as well.

3.5.3 Parsons problems

We looked for evidence in our analysis that Parsons problems were solvable with a reasonable amount of effort, but still challenging. We also searched for clues as to how we might improve Parsons problems, e.g., provide hints if they were challenging, or make them more challenging if they became too easy.

We concentrated our analysis on chapter four since it was the only chapter containing Parsons problems. Across all 11 Parsons problems, the average percent of people that eventually achieved a correct solution was 96.5% (standard deviation of 4.3). The lowest percentage for a correct solution was students in the High School course on problem 8 (83.6%). This is not surprising -- problem 8 is the first Parsons problem that requires indention. After the observational study of teachers using the ebook, our hypothesis was that Parsons problems that require indention would present a greater challenge to the students. This also confirms prior research, which showed that Parsons problems were easier to solve if they gave the structure of the solution (the number of lines and indention) (Denny et al., 2008). Figure 15 shows problem 8 with feedback telling the user that the highlighted block is not indented correctly. In Python, a code block is indicated by indention so all code in the body of a loop must be indented. Problems 8-11 all require the user to indent lines in the body of the loop correctly and these were the problems that required the most attempts before getting the problem correct as shown in Table 4.

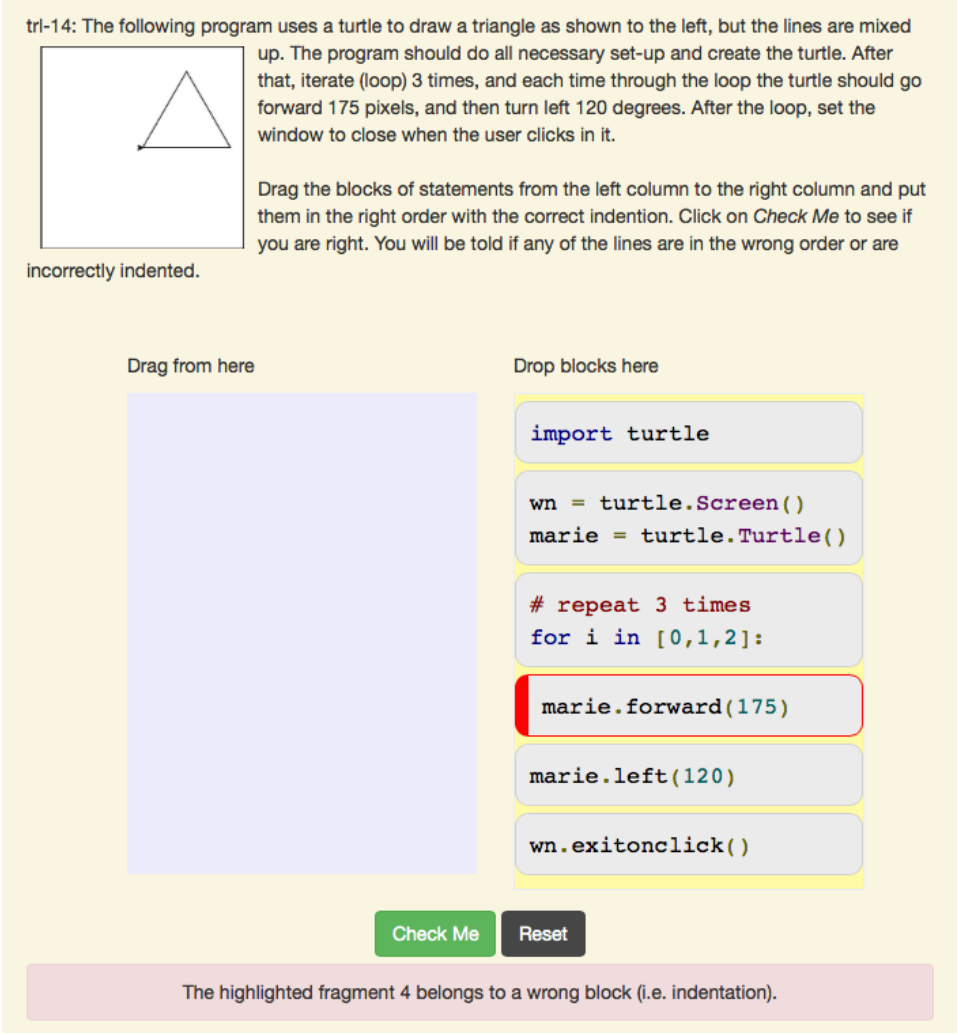


Figure 15. Parsons problem #8 with feedback that the indentation is wrong

Most students correctly solved most of the Parsons problems after only a couple of attempts. Table 4 shows the number of attempts it took for 50% and 75% of the people to get the problem correct, disaggregated by each Parsons problem. As you can see from this table the last four Parsons problems were the ones that students had the most difficulty solving. This is not surprising since these four problems all require indention and our observational study had shown that problems that required the correct indention were more difficult.

Table 4. Number of tries to correct for Parsons problems

Problem	# Blocks	Number of attempts for 50% of students to solve	Number of attempts for 75% of students to solve	Maximum attempts to the correct solution
1	3	1	2	42
2	5	1	2	25
3	5	1	1	38
4	5	1	1	29
5	5	1	2	23
6	5	1	2	28
7	6	1	1	24
8	6	4	7	80
9	6	1	3	46
10	7	2	9	109
11	8	2	4	76

Given that the number of permutations for a Parsons problem with 3 blocks is 6 (since statement order matters) and the maximum number of tries for problem 1 was well over 6 it is obvious that the student repeated incorrect solutions.

It is also interesting to look at the number of attempts users made before giving up on the Parsons problem – those who failed to ever solve the Parsons problem correctly.

Table 5 shows the number of attempts before 50% and 75% of the students quit trying to solve each Parsons problem, the maximum number of tries, and the total number of students who quit trying to solve that problem. Again, the number of tries that individuals made before quitting is more than expected with a maximum of 183 for problem 10. In comparison, the maximum number of attempts before correctly solving that problem was 109.

Table 5. Number of tries before quitting on Parsons problems

Problem	Number of attempts when 50% of students quit	Number of attempts when 75% of students quit	Maximum attempts before quitting	The number of students who quit / the number who attempted (% quit)
1	2	4	16	151 / 2087 (7.2%)
2	2	5	18	98 / 1866 (5.3%)
3	2	6	76	66 / 1726 (3.8%)
4	4	5	45	25 / 1443 (1.7%)
5	4	7	41	38 / 1348 (2.8%)
6	2	7	33	28 / 1302 (2.2%)
7	3	8	18	18 / 1177 (1.5%)
8	8	15	79	152 /1349 (11.3%)
9	6	12	69	101 / 1194 (8.5%)
10	9	17	183	183 / 1181 (15.5%)
11	7	13	72	105 / 995 (10.6%)

Figure 16 takes a closer look at the number of attempts before giving up on problem 10, which appears to have been the most difficult problem since it had the highest percentage of students quit without solving it.

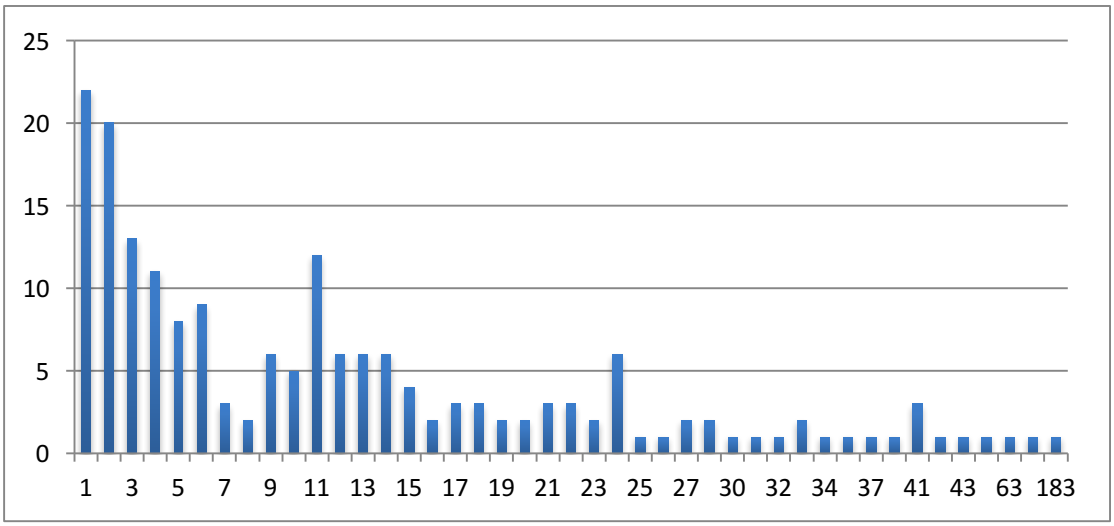


Figure 16. The number who quit after each number of attempts for problem 10. Note that the horizontal scale is not uniform.

The largest number of students quit after just one solution attempt, the second largest number after two attempts, and the third largest after three attempts. There are also spikes in the number of students who quit after 11 and 24 attempts. This information was useful later when deciding the number of attempts before activating the help on the intra-problem adaptive Parsons problems. We decided to allow help after three attempts at a solution since we didn't want to encourage students to overuse the help, but did want to provide help before too many students had quit trying to solve the problem.

3.6 Use Across All Features

One of the most interesting explorations of the data was to look at all the features within a specific chapter, chapter 4 (of the *How to Think Like a Computer Scientist – Interactive Edition*). We analyzed this data for the open access use of the ebook as well as the four courses. The results for the open access use of the book are shown in Figure 17, though the results for the four courses were similar. Each bar indicates the number of

unique users that performed each of the possible actions in chapter 4. Table 6 explains the labels and colors used in Figure 17. The x axis shows the number of people who did each action and the y axis is the order of the items in the chapter.

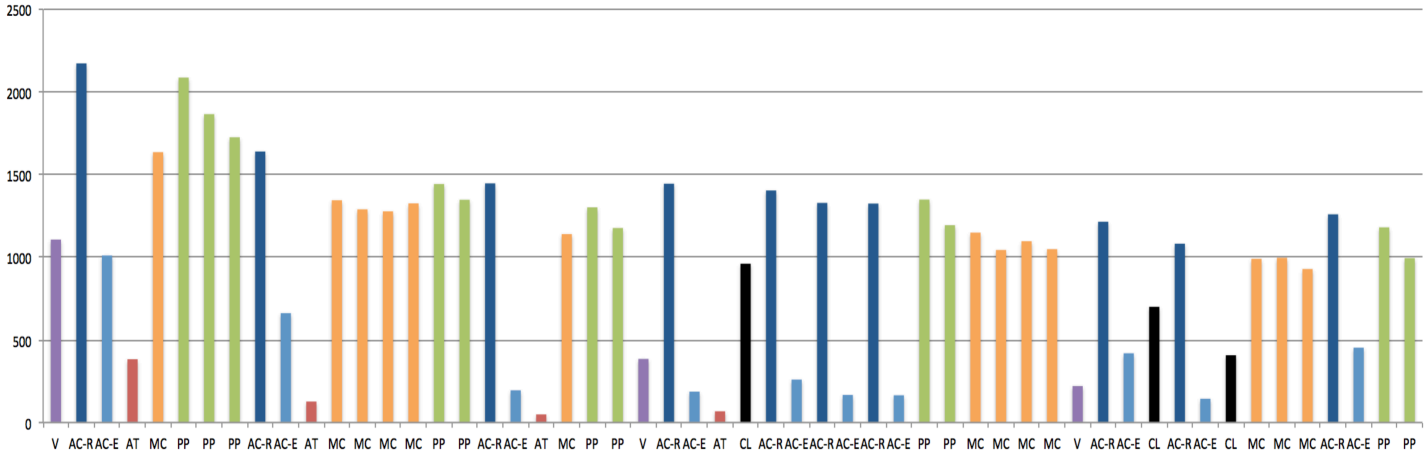


Figure 17. The number of unique users that did each action in chapter 4

Table 6. Explanation of labels and colors for Figure 17

Label	**Color**	**Explanation**
AC-E	Light Blue	Editing an Active Code example
AC-R	Dark Blue	Running an Active Code
AT	Red	Playing an audio tour
CL	Black	Clicking forward on a Code Lens
MC	Orange	Checking a multiple-choice answer
PP	Green	Checking a Parsons Problem
V	Purple	Playing a video

In general, we see a drop off in the number of people who perform each activity over the course of the chapter. By the end of the chapter, students do less of *everything*. However, it is important to note that they do *more* of the lower cognitive load practice

activities (e.g., multiple-choice questions and Parsons problems) than the higher cognitive load (and more traditional) computer science practice activity, editing code. This is interesting because readers were explicitly told to modify the code after the first, second, and last Active Code examples, but less than half of the students did this.

We see this same drop off in activity in each of the four courses we analyzed. One possible reason for this drop off is user fatigue. The entire chapter in this version of the ebook was one long HTML page. In the current version of this ebook, the long chapters have now been broken into much smaller sections. Another possible explanation for the drop off in activity is that students felt that they had a good grasp on the material and did not need the additional practice. However, students often overestimate the amount they have learned (R. A. Bjork et al., 2013).

One exciting finding was that more students attempted to solve the Parsons problems than the nearby multiple-choice questions. See Figure 18 for an example multiple-choice question from this chapter. See Figure 19 for an example nearby Parsons problem.

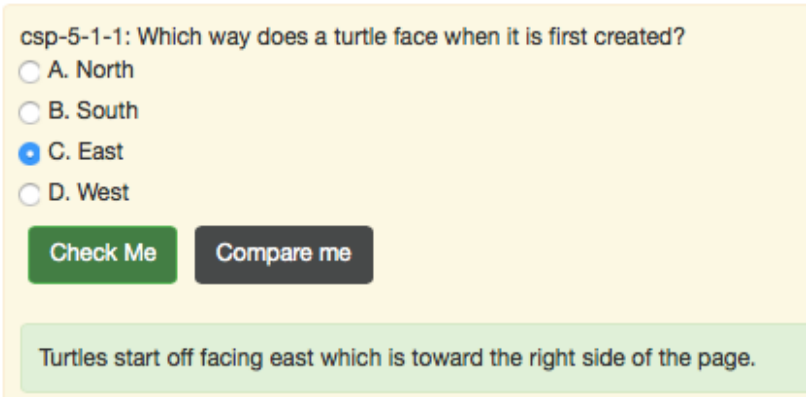


Figure 18. The first multiple-choice question from Chapter 4

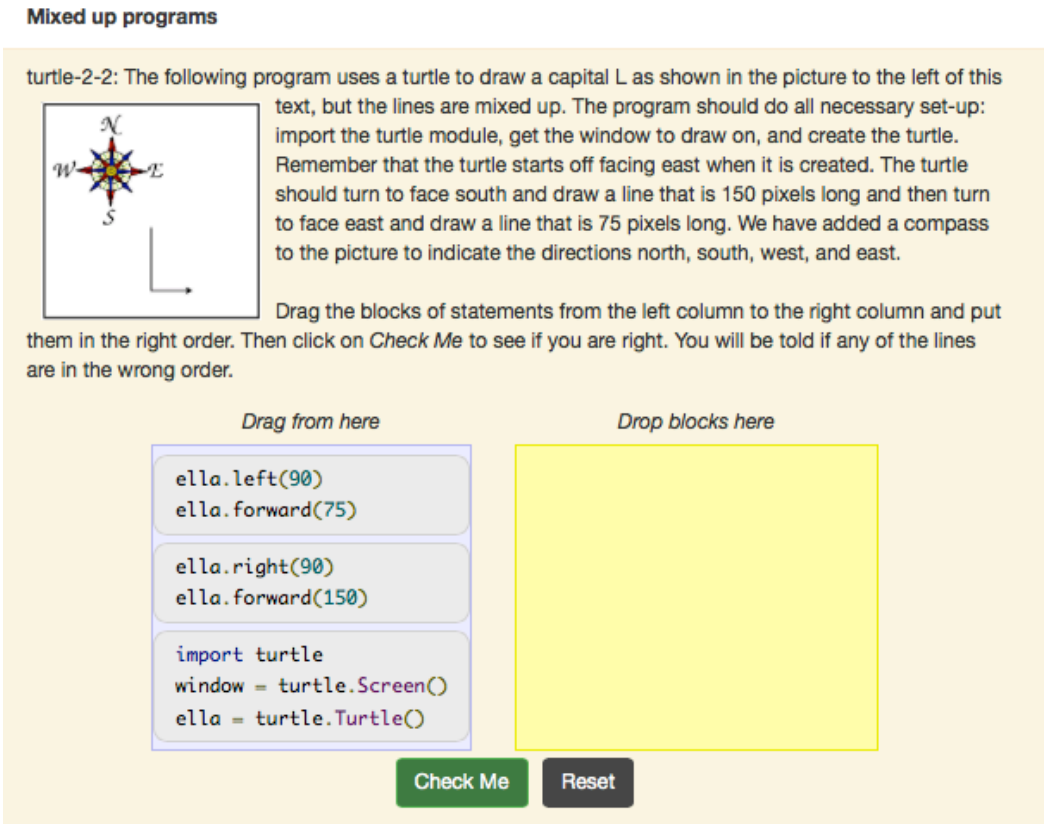


Figure 19. The first Parsons problem from chapter 4

The activities performed by the largest number of students were running Active Code examples, solving Parsons problems, and answering multiple-choice questions. The

activities that had the highest percent of continuation from first to last are running Active Code examples, answering multiple-choice questions, solving Parsons problems, and editing Active Code examples. Table 7 gives a further breakdown of these data for the open access use of the ebook.

Table 7. The number of people who did the first and last of each activity and the percentage of last versus first.

Activity	**# 1st**	**# Last**	**%last / first**
Active Code–run	2173	1260	58%
Multiple-choice	1636	929	57%
Parsons	2087	995	48%
Active Code-edit	1011	453	45%
Code Lens – fwd	961	407	42%
Video	1107	221	20%
Audio tour	383	68	18%

One of the interesting results of the data analysis is that the videos were not viewed as often as expected. Videos are the primary method for presenting information in many MOOCs and often are where learners spend a great deal of time in a MOOC (Seaton, Bergner, Chuang, Mitros, & Pritchard, 2014). However, only 51% of the people who ran the first Active Code example (2,173) in the chapter also watched the first video (1107) (see Table 7). By the third video in the chapter only 18% of the people watched the video (221) compared to the number that ran the next Active Code example (1,215).

Perhaps the students in the four courses we analyzed were not likely to watch the videos, because they also had face-to-face lectures. This may also be the case with the open access use of the book, but while some people use the open access book in their course we would expect that quite a few of the 2000 unique users were working on their

own rather than participating in a course. We anticipated that students working on their own would have viewed the videos much like MOOC students do. Another possibility for the low usage of the video feature is that the videos were not engaging. The videos in the ebook were screencasts, not the type of high quality videos found in MOOCs. From our teacher observations, we learned that at least some teachers stopped watching the videos when they realized that the videos covered the same material as the text. Finally, the videos may not have been played due to the confusing interface that required two clicks to play a video.

The audio tours also had very low usage rates. Only 383 people played the first audio tour compared to 2,173 people that ran the first Active Code example. One possibility is that the user interface made it difficult for people to notice the audio tours and thus they did not use that feature. This possibility is supported by the comments from the teachers in the observational study.

3.7 Conclusion

Our research questions were 1) *would readers use the new ebook features*, 2) *would some features be used more than others,* and 3) *were the practice problems at the right level of difficulty*? All of the new interactive features in the ebook were used, but some features were used much more than others. The teachers solved all of the practice problems in one or two tries, but the students had considerably more difficulty with some of the Parsons problems. This may be due to the fact that the teachers had more prior programming experience than the students. All of the teachers had been teaching

programming using drag and drop environments and one had taken a college-level programming course.

Both students and teachers ran the code examples, attempted the multiple-choice questions, and attempted the Parsons problems. Teachers did *all* of these activities in chapter four, but students did more of these at the beginning of the chapter four and fewer by the end of the chapter. Students show this same pattern of reduced use of the interactive features over the course of a chapter in all of the first nine chapters of the ebook. Teachers edited the code examples when instructed to by the ebook, but less than half of the students did this. Teachers also interacted with all of the Code Lens examples, but less than half of the students used this feature. Both teachers and students made very little use of the videos and audio tours. The observational study suggested that the teachers thought that the videos were not necessary since they covered the same material as the text. The observational study also suggested that the user interface made it hard to notice the audio tours. We changed the user interface to highlight the audio tours and also added text in a new ebook for teachers to encourage the teachers to try the audio tours. However, few teachers used the audio tours in a subsequent study (B. Ericson, Rogers, Parker, Morrison, & Guzdial, 2016).

An important finding is that more students attempted to solve the Parsons problems than tried to solve the multiple-choice questions after a worked example. This finding is a contribution to the research on Parsons problems since prior studies of Parsons problems [15, 23, 32] hadn't compared the use of Parsons problems with other

types of practice problems. However, this preference for Parsons problems may be due to a novelty affect if the students had not seen this type of problem before.

Parsons and Haden suggested that Parsons problems are at the appropriate level of difficulty if they are solvable by all students in two to three tries at most (Parsons & Haden, 2006). They did not explain their reasoning for this, but it is likely that they wanted the problems to not be too difficult and frustrating. However, we feel that the standard should be if 75% of the students solve the problems in two to three attempts, since some students give up quickly. This standard indicates that three of the 11 Parson problems may be too easy, since 75% of the students solved them in just one attempt. Five of the Parsons problems appear to be at the desired level of difficulty, since 75% of the students solved them in 2 or 3 tries. However, three of the Parsons problems took more than 3 tries for 75% of the students to solve them, so these might be considered too difficult without additional help. One surprise was the number of attempts that students made before getting the problems correct. It took from 23 to 109 tries for at least one student to get each problem correct. The data on how many attempts it took before students gave up on solving Parsons problems was useful in determining when to offer additional help. In intra-problem adaptation, help is available after three full attempts in order to prevent students from overusing the help, while still providing help before too many students quit trying to solve the problem.

At the ITiCSE 2013 conference, a working group developed a set of requirements and design strategies for ebooks (Korhonen et al., 2013). One of the top features on the "must include" list was the "editing and execution of code segments." Our results

suggest that code editing was not actually used much by the students, as compared to other activities. Low cognitive load activities like Parsons problems were much more commonly used. We believe that it's important for ebooks to support editing code. They might be perceived as inauthentic without code editing (Shaffer & Resnick, 1999). However, features that would actually be *used* and be *useful for learning* may not be what we might expect. Other kinds of activities may increase opportunities for engagement, while also contributing to learning.

Research in educational psychology suggests that a worked example plus interleaved practice problem approach leads to effective and efficient learning. This paper contributes to the research on worked examples plus practice by showing that the four teachers interacted with all of the worked examples and practice problems, however future research will have to determine if that interaction led to learning gains. While students didn't use all of the worked examples and practice problems, more of them attempted the low cognitive load practice problems than the higher cognitive task of code editing, even though they were instructed to edit the code in the ebook. These studies demonstrate that Parsons problems are a promising type of practice problem, since more students attempted Parson problems than attempted the multiple-choice questions after a worked example.

Since teachers found the Parsons problems too easy, but some of the students found the problems difficult, I conceived the idea of making Parsons problems dynamically adaptive in order to keep the difficulty of the problem in the learner's zone of proximal development, provide desirable difficulty, and provide deliberate practice.

However, I first wanted to test my hypothesis that students who solved non-adaptive Parsons problems with paired distractor would learn as much in less time than those who fixed code with the same errors as the distractors and those who wrote the equivalent code. That study is described in the next chapter.

CHAPTER 4. SOLVING PARSONS PROBLEMS VERSUS FIXING AND WRITING CODE

The log file study provided evidence that more students attempted Parsons problems than nearby multiple-choice questions (B. J. Ericson et al., 2015), which indicated that students found Parsons problems engaging. In addition, teachers who used a new interactive ebook that our research group created for the Advanced Placement (AP) Computer Science Principles (CSP) course rated Parsons problems as valuable at twice the rate of multiple-choice questions or fill in the blank questions in online feedback (B. Ericson et al., 2016). This indicates that teachers also found solving Parsons problems helpful. Both results encouraged me to study Parsons problems in more depth. I next compared the efficiency and effectiveness of solving Parsons problems with distractors, versus fixing the same code with the same errors as the distractors, versus writing the equivalent code. The rest of this chapter is adapted from a paper that was presented at the Koli Calling International Conference on Computing Education Research (B. J. Ericson, Rick, & Margulieux, 2017).

4.1 Research Question

RQ1: What is the efficiency (time to complete practice problems), effectiveness (learning gains from pretest to posttests), and cognitive load of 1) solving non-adaptive Parsons problems with distractors versus 2) fixing the same code with the same errors as the distractors versus 3) writing the equivalent code?

4.2 Goals for Study

While several researchers have hypothesized that solving Parsons problems could result in more efficient learning than writing the equivalent code (Denny et al., 2008; Parsons & Haden, 2006), none to our knowledge had empirically tested this assumption. Some researchers have found a notable correlation between scores on Parsons problems and performance on different write code problems (Cheng & Harrington, 2017; Denny et al., 2008), however these studies have not compared groups solving the same problems. In addition, no researchers had compared solving two-dimensional Parsons problems with paired distractors to fixing code with the same errors as the distractors. Since the learner doesn't have to type the code while solving either Parsons problems or fix code problems, they might have similar completion times.

The purpose of this study was to investigate the efficiency, effectiveness, and cognitive load of learning from solving two-dimensional Parsons problems with paired distractors, versus fixing code with the same distractors as errors, versus writing the equivalent code.

Our hypotheses were as follows.

- **H1A:** Learners who solve Parsons problems will finish the instructional problems faster than the learners who fix or write code.
- **H1B:** Learners who solve non-adaptive Parsons problems with distractors will achieve similar learning gains from pretest to immediate posttest and delayed posttest than learners who fix code

with the same errors as the distractors or learners who write the equivalent code.

- **H1C:** Learners will report lower cognitive load after solving Parsons problems versus learners who write or fix code.

4.3 Study Design

This was a between-subjects design, with one pretest and two posttests. There were two sessions in the study. The first session was 2.5 hours and included consent, a demographic survey, pretest, instructional material, a cognitive load survey, and a posttest. The second posttest, which lasted one hour and was held one week later, was administered to measure retention of the instructional material.

The instructional material in the first session contained four worked example and practice pairs. Students were randomly assigned to one of three practice conditions for the instructional material: 1) solving two-dimensional Parsons problems with paired distractors, 2) fixing code with the same errors as the distractors, or 3) writing the equivalent code. The instructional practice condition was the independent variable. The dependent variables were the performance on the pretest and posttests, the time spent on each practice problem, and the cognitive load survey results.

4.4 Study Procedures

This study consisted of two separate sessions one week apart. Both sessions were held in a closed classroom with all participants attending at the same time. Students were instructed to bring their laptops, and were provided with scratch paper and a pen. All of

the study materials were online and students were asked to only use those materials, even though they had access to the Internet. Proctors checked that the students were on task and not visiting other web sites. The scratch papers were collected and analyzed to replicate a previous study (Lister et al., 2004) of code tracing on paper (Cunningham, Blanchard, Ericson, & Guzdial, 2017).

In the first session, the procedure was 1) provide consent and randomly be placed into one of the three practice conditions, 2) complete the demographic survey, 3) complete familiarization activities (practice using the environment), 4) complete the pretest, 5) review material on lists and ranges, 6) complete four worked example plus practice pairs, where the type of practice problem differed based on the condition, 7) complete a cognitive load survey, and 8) complete the immediate posttest. The procedure is shown in Figure 20.

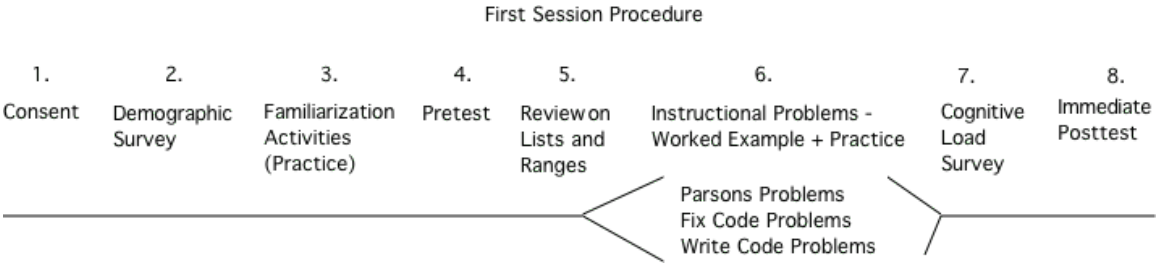


Figure 20. The first session procedure

At the second session, a week later, participants completed the second posttest, which was isomorphic to the first posttest. Only the variable names and some values were changed, but the structure of the problems was the same, meaning that they required near transfer to solve. Near transfer is being able to solve a new problem in a similar context

to one that you have already solved. The second posttest tested for retention of the material one week later.

4.5 Study Materials

We developed, tested, and refined our materials through observations of three undergraduate students from an introductory computing course for computer science majors. Each student was observed as he or she worked through the material for one of the three conditions. After the observational study, we added more familiarization material, because some of the students had difficulty using the environment.

I next conducted a pilot study with 24 undergraduate students from an introductory course for computer science majors. In the pilot study, five (21%) of the 24 students submitted at least one solution to the pretest Parsons problem that contained both a correct block and its paired distractor. This indicated that they didn't realize that each distractor was shown paired with the correct code, for at least some of the distractors. At that time, the distractors were shown either above or below the correct code, but there was no other visual indication that they were paired.

After the pilot study, we added the purple edge decorations shown in Figure 21 to better indicate that each distractor block was displayed paired with its correct code block. The distractor blocks would randomly be placed either above or below the correct block. We also added the same subgoal label comment to both the correct and distractor code block to further indicate that the blocks were paired. The blocks in the source area were always displayed with the purple edge decorations, which helped to show that they were one of a pair. However, the purple edge decorations were not shown on a block after it

was dragged to the solution area on the right. If a paired block was dragged into the solution area, and then later dragged back to the source area on the left, it would automatically move to be randomly displayed either above or below the paired block and the purple edge decorations would be added again to show the pair of correct and incorrect blocks.

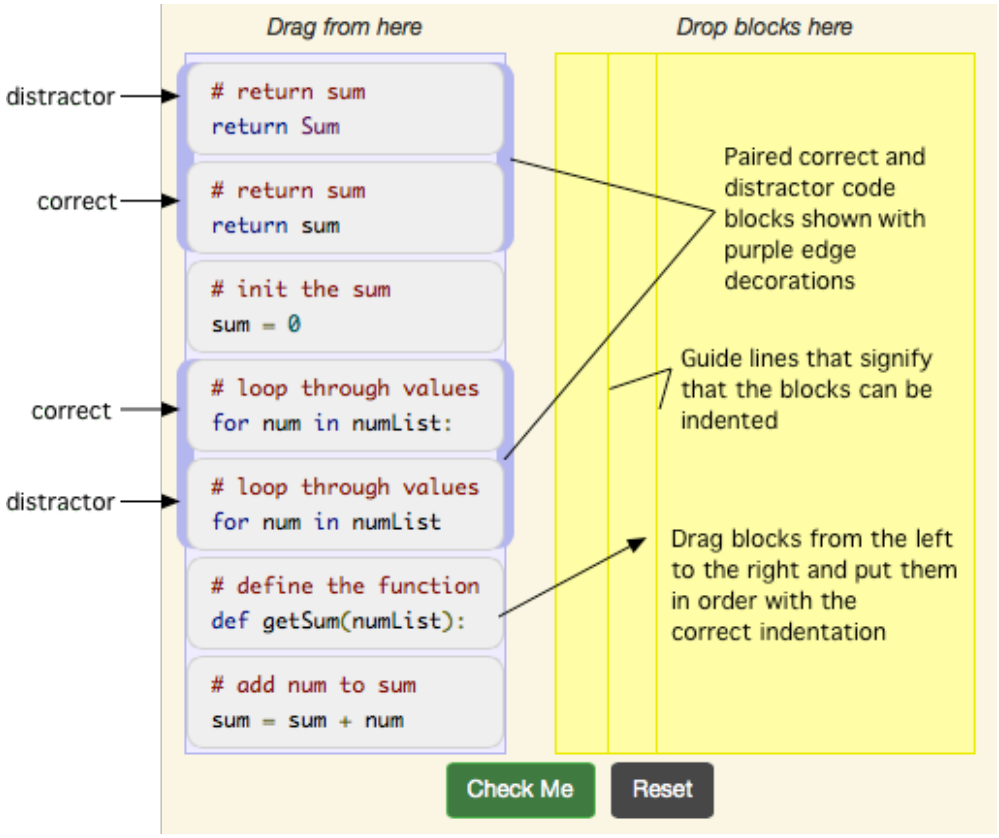


Figure 21. A 2d Parsons Problem with Paired Distractor and Correct Blocks

4.5.1 Demographic Survey

The demographic survey asked for the participant's age, gender, race, first spoken language, comfort level with reading English, high school grade point average, college grade point average, current major, expected grade in the course, and prior programming

experience. If they had any prior programming experience, they were also asked what courses and where they took them and how many years they had been programming. In addition, participants were asked to rate their ability to read, fix, and write Python code on a 5-point Likert scale.

4.5.2 Familiarization (Practice) Material

The familiarization activities included instruction on how to start and finish a timed exam, how to get to the next page, how to answer multiple-choice questions, how to check the solution for the fix code and write code problems, and how to drag blocks and check the solution on a Parsons problem.

The familiarization (practice) section also included two easy practice multiple-choice questions, a practice fix code problem with instructions for how to fix the problem, a practice Parsons problem and a write code problem. Both the fix code problem and the Parsons problem displayed the correct solution above the problem, as the goal for this section was for the students to learn how to answer each type of question, rather than to test their ability to create a correct solution.

4.5.3 Pretest

There were four timed exams in the pretest. The participants had 15 minutes to complete the first timed exam of five multiple-choice questions and 10 minutes to complete each of the other three timed exams (a fix code problem, a Parsons problem, and a write code problem). It was important to include each of the types of problems in the three conditions to prevent any of the conditions from having an advantage in gains

from pretest to posttest. I included multiple-choice questions to leverage prior research from undergraduate students in a first programming course for majors.

The five multiple-choice questions required tracing code with lists, ranges, selection, and iteration. The questions included code to find the minimum value in a list between a range of indices, return the count of the number of times a target value appeared in a range of indices in a list, trace the values of variables in a complex for loop, and return the average of values in a range of indices in a list (as shown in Figure 22).

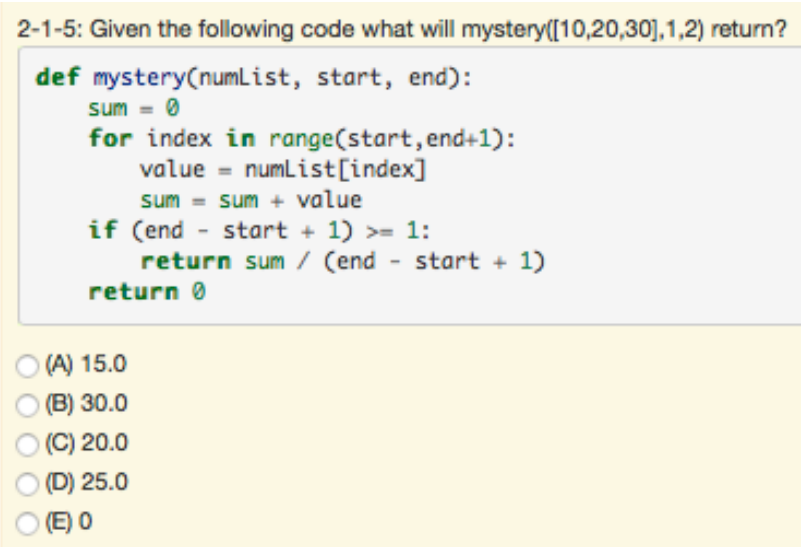


Figure 22. One of the pretest multiple-choice questions

One multiple-choice question provided code that was intended to return the longest run in a list of numbers, but the code contained an error and the student had to select the answer that matched what the code actually returned.

The second timed exam contained one fix code problem. It was a modified version of Soloway's rainfall problem, which has been extensively studied (Fisler, 2014; Simon, 2013; Soloway, 1986) as shown in Figure 23. This problem totals all of the non-

negative values in an input loop until a sentinel value is reached and then outputs the average. The solution should also avoid a division by zero. The problem was modified to loop through a list of numbers rather than read input until a sentinel value was reached. Simon found that students still perform poorly on this problem and that students are not used to reading input in a loop until a sentinel value is reached (Simon, 2013). The instructions explained the algorithm in English, provided example input and output, and provided hidden unit tests.

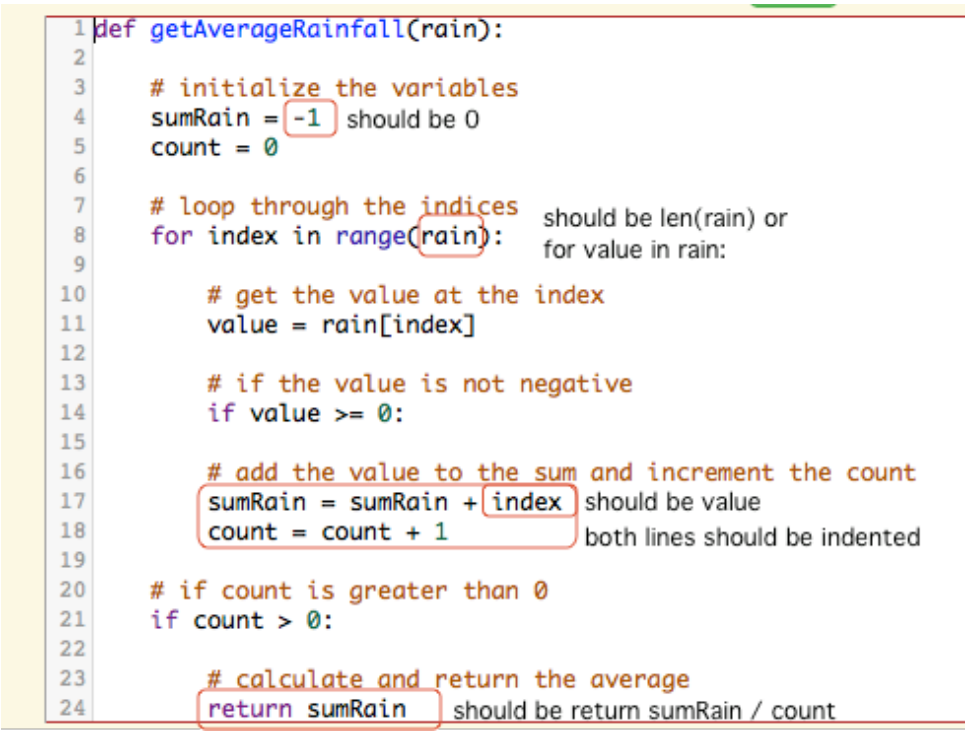


Figure 23. The pretest fix code problem with errors highlighted

The third timed exam contained one Parsons problem to create a function to calculate and return the average of the values at a range of indices (inclusive) in a list. This problem was isomorphic with the multiple-choice question shown in Figure 22. The problem had five paired distractors as shown in Figure 24. The instructions explained the

algorithm in English, provided example input and output, and gave feedback on the solution. The feedback was either that the solution was correct, or too short, or one or more code blocks were either out of order or the wrong blocks (and these blocks were highlighted in red), or that the indentation was wrong (and yellow decorations were added to the side of the block with arrows to indicate the direction the block needed to move).

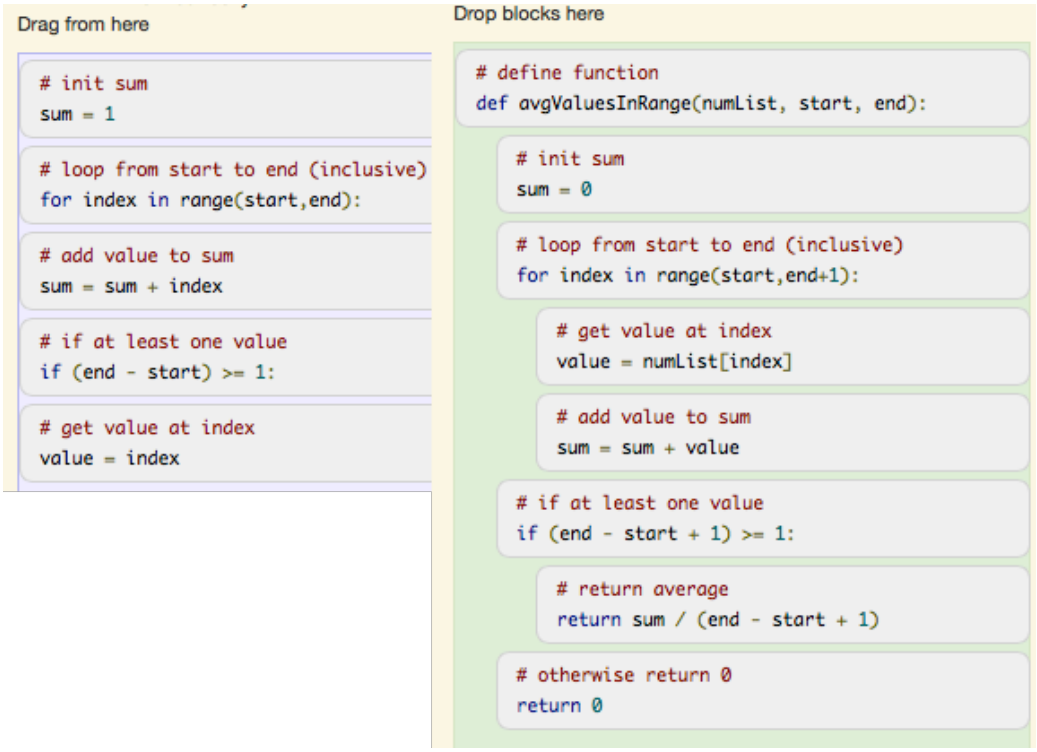


Figure 24. The pretest Parson problem showing unused distractors on the left and the correct solution on the right.

The fourth timed exam contained one write code problem as shown in Figure 25. This problem asked the participant to write a method to check if a trail was level between a start and end index (inclusive). A trail was considered to be level if the difference between the minimum and maximum values was less than or equal to 10. The problem

provided the function header and hidden unit tests. The instructions explained the algorithm in English, provided example input and output, and provided hidden unit tests to test the solution.

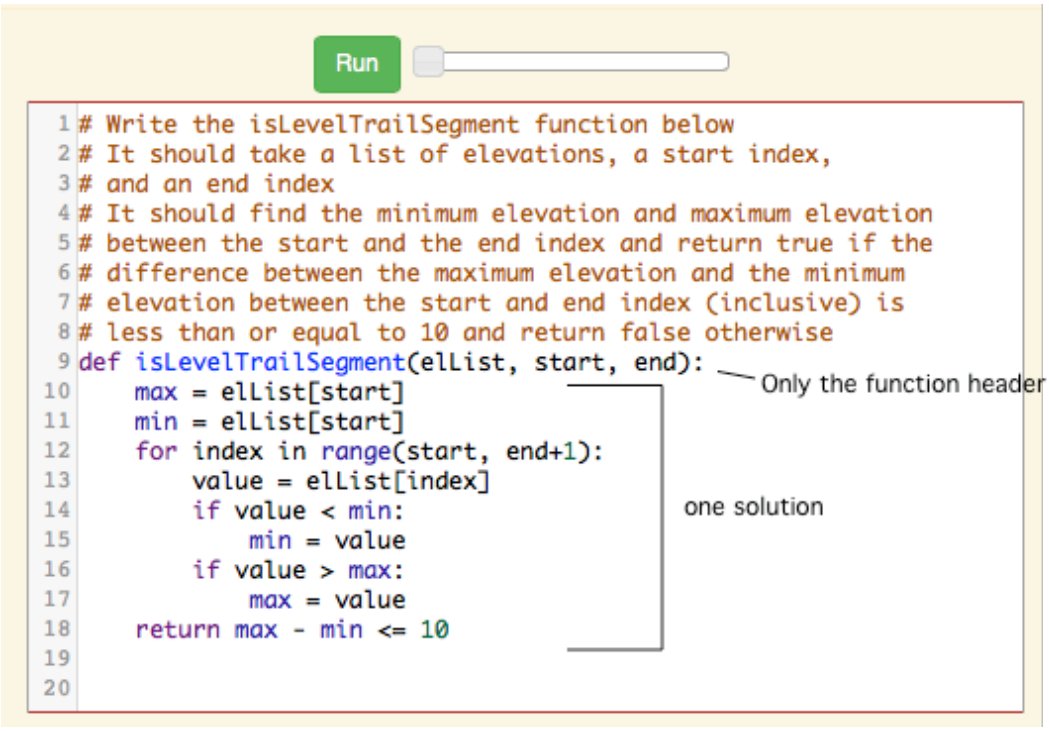


Figure 25. A correct solution to the write code problem

4.5.4 Review Material

The review material explained what a list was, how to use the range function to create a list, how to get a value from a list, how to get the length of a list, how to loop through all values in a list, and how to loop through a range of indices in a list. It contained example Python code that the participant could run. The students in the experiment had already covered these concepts and had moved on to cover more advanced topics, so we felt it would be helpful to provide this review material.

4.5.5 Instructional Material

The instruction material contained four worked examples with interleaved practice problems. The worked examples contained an algorithm in English, example input and output, and runnable Python code with hidden unit tests, which all passed. The practice problems varied by condition with one group solving two-dimensional Parsons problems with paired distractors, one solving fix code problems with the same distractors as errors, and the third writing the equivalent code. Each of the practice problems also contained an algorithm in English, example input and output, and a way to test the solution. Each practice problem was in a timed exam and each had a time limit of 10 minutes. The page following the timed exam displayed an English description of a correct solution and the code for that solution.

The first worked example returned a count of the number of times a target value appeared in a list using a loop that looped through all the indices. The associated practice question was to return the count of a target value in a given range of indices (inclusive). The second worked example returned the maximum value from a list and the associated practice problem was to return the minimum value. The third worked example returned the average of the values in a list and protected against a divide by zero error. The associated practice problem returned the average, but didn't include the lowest value in the list in the average and also guarded against a divide by zero error. The fourth worked example returned the minimum value in a given range of indices (inclusive). The associated practice problem returned the maximum value in a given range of indices (inclusive).

4.5.6 Cognitive Load Survey

To measure the cognitive load for each of the practice conditions we used the CS Cognitive Load Component Survey, which had been tested and shown some initial validation in computer science (Morrison, Dorn, & Guzdial, 2014). This survey was adapted from the Cognitive Load Component Survey that has been used to measure cognitive load in statistics and health sciences (Leppink, Paas, Vleuten, Gog, & Merriënboer, 2013).

4.5.7 Posttests

The immediate posttest in the first session had the same questions as the pretest. The second posttest, which was administered one week later, was isomorphic to the first posttest, meaning that the problems to be solved had the same structure, but different surface level features, like variable names.

4.6 Participants

Undergraduate students were recruited from two sections of a first computer science course for computing majors at the Georgia Institute of Technology, a research-intensive university in the United States. The sections had different instructors, but they followed the same curriculum with the same homework and assessments. This course covers introductory programming concepts in Python including variables, selection, iteration, and lists. At the time of the study the course had covered all of these topics and was covering files and dictionaries. I visited the course during lecture to recruit participants and also sent an announcement to all of the students enrolled in the course.

Participants could earn 2.5 points of extra credit for completing the first session and another 2.5 points of extra credit for completing the second session one week later. Students who did not participate in the pilot study or large-scale study could alternatively earn up to 5 points of extra credit by writing a paper on a computing innovation, which I graded and that grade was submitted to the course instructors. I was not involved in the teaching of the course.

4.7 Analysis

A total of 159 students participated in the first session. However, 24 of these students did not answer at least one question during the session or spent less than 30 seconds answering a question. I am reporting on the data from 135 students (45 in the fix condition, 44 in the Parsons condition, and 46 in the write condition) from the first session. Students were not required to come back for the second session one week later, but earned an additional 2.5 points of extra credit for completing that session. A total of 106 students returned for the second session. Of these, 82 completed all the questions in both the first session and second session and spent at least 30 seconds on each question (27 in the fix condition, 25 in the Parsons condition, and 30 in the write condition).

4.7.1 Time and Score Data Preparation

For each instructional practice problem, the elapsed time in seconds was calculated from the start and end time to compare the efficiency of the three conditions. I created grading rubrics for the write and fix code problems on the pretest and posttests. Two people graded each problem independently and then met to resolve any differences

in scores. The hand graded scores on the fix and write problems correlated with the number of unit tests passed ($p < .001$ for all). A unit test checks that the expected output from a function matches the actual output. The fix and write code problems had five to six unit tests each. The student was shown the input to the unit tests, the expected output, the actual output, and whether each test passed or failed.

I automated the grading for the Parsons problems. Grading started from the beginning of the solution and each correct line in the proper order received one point and if the line or its paired distractor was indented correctly it received half a point. Grading continued until a line was found that was neither the correct line nor its paired distractor (i.e. a line out of order). Grading then continued from the end of the solution in the same fashion toward the first line that had been found to be incorrect. I also reviewed the middle of the solutions manually to give credit if at least two consecutive lines were in the correct order relative to each other. This grading approach was based on my observation that learners had the most difficulty in the middle of the solution. I also wanted the grading to be similar to the grading of the fix code problems, and the fix code problems had the advantage that the code was already in the correct order.

The data was checked for normal distribution using skewness, whether the peak of the bell curve is in the middle, and kurtosis, whether the bell curve is too narrow or wide (Gravetter & Wallnau, 2016). For all pretest measurements, skewness and kurtosis checks were within the acceptable +/-2 range. For all posttest measurements, skewness was about -2, meaning that there was a slight negative skew (i.e., bell curve looks like it is leaning towards the larger numbers), but these values were still acceptable. Kurtosis,

however, was above 3 in all cases, meaning that the scores clustered more closely around the mean than in a normal distribution. Based on these results, there was probably a slight ceiling effect for the posttests in which many participants scored the highest score possible. Most parametric statistical tests, including all of those that were used, are robust to abnormal kurtosis, meaning that they are still valid this type of distribution. Parametric tests were therefore used to analyze the results instead of their non-parametric equivalents, which tend to be more conservative with lower statistical power (Trochim & Donnelly, 2006).

4.7.2 Testing for Efficiency

The Parsons problem condition had the lowest average completion time for each of the four practice problems as shown in Table 8. There was a significant difference between the conditions on the practice problem completion time as measured by an independent measures one-way analysis of variance (ANOVA) $F(2,133) = 10.835$, $p < 0.001$. A Least Significant Difference (LSD) post-hoc test indicated that students in the Parsons problem condition took significantly less time to finish the four practice problems than students in the fix code ($p<0.001$) and write code conditions ($p<0.001$). However, there was no significant difference in completion time between the write code and fix code conditions. While one may have thought that a fix code problem could have a similar completion time to a Parsons problem since neither require the student to type all the code, the difficulty that novice students have with understanding complier errors and debugging code likely contribute to the longer completion times for fix code problems.

Table 8. Mean Time in Seconds (and Standard Deviation) to Complete each Practice Problem by Condition

	Prac. 1	***Prac. 2***	***Prac. 3***	***Prac. 4***	***Total***
Parsons	84.20 (34.77)	83.64 (35.99)	227.42 (124.66)	77.98 (41.29)	473.24
Fix	114.49 (79.17)	147.67 (128.32)	313.42 (153.40)	103.91 (65.67)	679.49
Write	171.63 (137.61)	113.13 (98.62)	313.65 (153.33)	115.54 (69.28)	713.96

4.7.3 Testing for Effectiveness

The pretest measures (multiple-choice, fix, Parsons, and write) were condensed into a single composite pretest score. To ensure that this was valid and that all of the pretest measures were measuring the same underlying construct, factor analysis was used with varimax rotation. The analysis showed that the four tests loaded onto one factor, which we will call prior knowledge, based on the scree plot and eigenvalues. The factor loadings for each of the individual tests was above .7, the typical cutoff: fix score = .75, write score = .85, multiple-choice score = .76, and order score = .79.

None of the practice conditions performed better than the other conditions on the immediate posttest measurements (multiple-choice, fix, Parsons, or write). The mean and standard deviation by condition is shown in Table 9. No interactions between condition and performance on the immediate posttest measures were found either, meaning that participants who practiced on Parsons problems performed as well on the writing problem in the immediate posttest as participants who practiced on writing problems and

vice versa. In addition, there was no significant difference by condition on performance on the delayed posttest (one week later). The mean and standard deviation by condition for the students who did the pretest, immediate posttest, and delayed posttest is shown in

Table 10. Remember that 135 students completed the pretest, instructional problems, and immediate posttest, while 82 students completed the pretest, instructional problems, immediate posttest, and delayed posttest one week later.

Table 9. Mean score (and standard deviation) by condition for those students (n=135) who completed the pretest and immediate posttest

Fix Condition (n = 44)	*Pretest (std dev)*	*Immediate Posttest (std dev)*
Multiple-Choice	3.48 (1.45)	3.50 (1.50)
Fix	10.36 (2.01)	11.41 (1.23)
Parsons	11.76 (1.01)	11.63 (1.48)
Write	9.50 (3.56)	10.18 (3.49)
Parsons Condition (n = 45)	*Pretest (std dev)*	*Posttest (std dev)*
Multiple-Choice	3.22 (1.17)	3.78 (1.17)
Fix	10.96 (1.69)	11.42 (1.34)
Parsons	11.61 (1.31)	11.77 (1.19)
Write	8.40 (3.68)	9.78 (3.27)
Write Condition (n = 46)	*Pretest (std dev)*	*Posttest (std dev)*
Multiple-Choice	3.41 (1.24)	3.72 (1.19)
Fix	10.96 (1.69)	11.37 (1.25)
Parsons	11.38 (1.93)	11.70 (1.28)
Write	9.48 (3.75)	10.30 (2.62)

Table 10. Mean score (and standard deviation) by condition for those students (n=82) who completed the pretest, immediate posttest, and delayed posttest

Fix Condition (n = 27)	*Pretest (std dev)*	*Immediate Posttest (std dev)*	*Delayed Posttest (std dev)*
Multiple-Choice	3.74 (1.29)	3.85 (1.23)	3.70 (1.54)
Fix	10.33 (2.17)	11.52 (1.01)	11.19 (1.33)
Parsons	11.78 (0.97)	11.56 (1.69)	11.54 (1.70)
Write	9.78 (3.34)	10.52 (3.12)	9.89 (3.43)
Parsons Condition (n = 25)	*Pretest (std dev)*	*Posttest (std dev)*	*Delayed Posttest (std dev)*
Multiple-Choice	3.16 (1.28)	3.64 (1.32)	3.4 (1.44)
Fix	10.84 (1.75)	11.48 (1.45)	11.48 (1.26)
Parsons	11.62 (1.36)	11.7 (1.5)	11.7 (1.5)
Write	9.08 (3.38)	10.04 (3.13)	10.08 (3.04)
Write Condition (n = 30)	*Pretest (std dev)*	*Posttest (std dev)*	*Delayed Posttest (std dev)*
Multiple-Choice	3.67 (0.96)	3.93 (0.94)	3.9 (1.35)
Fix	10.97 (1.71)	11.5 (1.04)	11.1 (1.73)
Parsons	11.4 (1.94)	11.63 (1.49)	11.15 (2.75)
Write	9.97 (3.26)	10.53 (2.30)	10.7 (2.59)

When analyzing repeated measures data, as we have for the pretest, immediate posttest, and delayed posttest, it is common to violate the assumption of sphericity, as tested with Mauchly's test. The data violated the sphericity assumption, $p < .001$, so the Huynh-Feldt correction was used to make the ANOVA results more conservative. There was a significant difference between the pretest and posttests using an omnibus repeated measures ANOVA for the fix problem F(using Huynh-Feldt correction; 1.9, 161.2) = 7.34, $p = .001$, and the write problem F(using Huynh-Feldt correction; 1.6, 139.9) = 4.56,

p = .018. Participants also performed better on the fix and write code problems on both posttests than on the pretest. However, their performance on these problems on the delayed posttest was worse than the immediate posttest, though not so bad as to be statistically equivalent to the pretest. There were no significant differences from the pretests to the posttests on the multiple-choice questions or the Parsons problem. For the multiple-choice questions this may have been due to the lack of feedback on the correctness of the pretest answers. It is possible that the students simply remembered what they had answered before and used the same answer, since they were not told if the answers were wrong. The lack of significant difference on the Parsons problem is likely due to a ceiling effect, because each group had a mean above 11 (out of 12 possible points).

4.8 Cognitive Load

There was no significant difference in the self-reported cognitive load measures between the three conditions, $F(2, 132) = 1.21$, $p = .30$. However, the students in the Parsons problem condition solved the same problems as those in the fix code and write code conditions in significantly less time as shown in Table 8, which indicates that Parsons problems may have a lower cognitive load than fixing or writing code.

4.9 Comparing Demographic Data to Performance

Of the 146 students who filled out the demographic survey at the first session, 66 (45%) identified as male, 79 (54%), as female and one (1%) as other. The self-reported race was 72 (49%) White, 59 (40%) Asian, 14 (10%) Black or African American, 8 (5%)

Hispanic, 1 (1%) Pacific Islander, and 1 (1%) Middle Eastern. Students could select more than one race. Most students spoke English as their first language, 112 (77%). Other first languages were Mandarin, Hindi, Spanish, Korean, Vietnamese, Serbian, Farsi, Japanese, Gujarati, Kannada, Thai, Tamil, Greek, and Chinese. The majority of the students were 18 as shown in Table 11, but two students were in their 30's.

Table 11. Number and percentage of students by age

Age	17	18	19	20	21	22	23-29	30-39
# & %	5 (3%)	97 (66%)	29 (20%)	5 (3%)	3 (2%)	2 (1%)	3 (2%)	2 (1%)

While the majority of the students were computer science majors, there were students from a variety of majors including engineering, math, science, and liberal arts as shown in Table 12.

Table 12. The number and percentage by major

Major	**Number (% of Total)**
Computer Science	84 (57%)
Engineering	14 (10%)
Mathematics	11 (8%)
Computational Media	10 (7%)
Business	10 (7%)
Biology	6 (4%)
Physics	5 (3%)
Biochemistry	3 (2%)
Literature, Media, and Comm.	2 (1%)
Economics	1 (1%)

To check for possible interaction between the demographic data and the conditions, we created a composite score using the fix and write code problem scores

from the two posttests. We found no interaction between condition and demographic characteristics that affected performance.

There was a moderate correlation for age, $r(88) = -.22$, $p = .04$, with younger students performing better than older students. There was also a moderate correlation by major with computer science majors, $\rho(88) = -.24$, $p = .02$, performing better than the non-computer science majors. Not surprisingly, there was a moderate correlation with prior experience, $\rho(88) = .24$, $p = .02$. There were moderate correlations on all of the self-reported measures of ability to read, $\rho(88) = .32$, $p = .002$; fix, $\rho(88) = .28$, $p = .008$; and write, $\rho(88) = .34$, $p = .001$, Python code. We found a strong correlation between expected grade in the course and performance. $\rho(88) = -.50$, $p < .001$. There were no significant correlations for the other demographic characteristics including race, gender, first language, high school grade point average, or college grade point average.

4.10 Discussion

My hypotheses were:

- **H1A:** Learners who solve Parsons problems will finish the instructional problems faster than the learners who fix or write code.
- **H1B:** Learners who solve non-adaptive Parsons problems with distractors will achieve similar learning gains from pretest to immediate posttest and delayed posttest than learners who fix code with the same errors as the distractors or learners who write the equivalent code.

- **H1C:** Learners will report lower cognitive load after solving Parsons problems versus learners who write or fix code.

The students in the non-adaptive Parsons problem condition completed the four instructional practice problems in significantly less time than those in the fix code or write code conditions. This supports **H1A**. There was no significant difference between the completion time for the students in the fix code or write code conditions. Fix code problems, like Parsons problems, have an advantage over write code problems, because the student doesn't need to type the code for the solution. However, Parsons problems with paired distractors appear to take less time for students to solve since they don't have to interpret compiler errors or debug code. They can simply pick between the paired correct and incorrect blocks.

There was a significant improvement from the pretest fix and write code problems to the same problems on the immediate posttest as well as on the posttest one week later, which provides evidence of near transfer and retention. These findings, coupled with the fact that there was no significant performance difference on the posttests by condition, support **H1B**.

There was no significant difference on the self-reported cognitive load survey by condition, so **H1C** was not supported. While the cognitive load survey that was used had been initially validated, it may not be an effective measure for comparing the cognitive load of different types of practice problems. The first study was 2.5 hours long and included many different parts, which were completed one after the other without a break. It is possible that students were responding to the difficulty of the entire study rather than

just the instructional section or were fatigued and just wanted to finish. Further studies should be done to test if the self-reported cognitive load of solving Parsons problems is lower than that of solving fix code and write code problems. A within-subjects study might be a better approach for testing self-reported cognitive load. However, students in the Parsons problem condition solved the same practice problems in significantly less time than those in the fix code or write code conditions, which implies that Parsons problems do have a lower cognitive load.

4.11 Limitations

It is possible that the performance improvements may not be solely due to the practice condition. Students may have learned from the pretest problems, review material, worked examples, or answers to the four practice problems. This study could have been improved by adding a control group that did an off-task activity rather than solve the four practice problems. This would have strengthened the claim that the performance gains were due to the practice problems, and not the other materials. The retention results on the delayed posttest could have also been partially due to learning in the students' course during the week after the immediate posttest, however the topics covered that week were more advanced. Further experiments should be done to verify that solving Parsons problems results in equivalent performance gains compared to fixing and/or writing code.

4.12 Conclusion

This study provided initial evidence that solving two-dimensional Parsons problems with paired distractors takes significantly less time than fixing the same code with the same errors as the distractors or than writing the equivalent code, while still resulting in statistically significant improvement in scores from pretest to immediate posttest and retention one week later. This implies that solving Parsons problems with paired distractors could be a more efficient, but just as effective, form of practice than writing or fixing code. However, it is possible that the learning gains were not solely due to the instructional practice problems. Students could have learned from the review, answers to the practice, or from repeated exposure to the same or similar problems. This study could have been improved by adding a control group that did off-task practice to test if the learning gains were due to the practice condition.

In an earlier log file analysis of students, I found that while most students solved each Parsons problem, some students clearly struggled to solve each problem, and some never solved each problem (B. J. Ericson et al., 2015). In the next experiment, the software was modified to allow for intra-problem adaptation. In intra-problem adaptation, if the user is struggling to solve the current problem we can make it easier by removing distractors, providing indentation, and combining blocks. We conducted an observational study of teachers solving both intra-problem adaptive Parsons problems and non-adaptive Parsons problems in order to test the effectiveness of our adaption process and teacher's perception of the effectiveness of solving Parsons problems in learning to fix and write code.

CHAPTER 5. TESTING THE EFFECTIVENESS OF INTRA-PROBLEM ADAPTIVE PARSONS PROBLEMS

While the teachers in my early teacher observation study discussed in chapter three, solved all of the Parsons problems and found them a bit too easy, a log file analysis of students solving the same problems showed that some students struggled to solve the same problems, and around 15% never solved one of the more difficult problems (B. J. Ericson et al., 2015). The cognitive load of a Parsons problem is based on the intrinsic difficulty of the problem, but also on the learner's prior knowledge. How do we help learners with less prior knowledge succeed on Parsons problems? It is important to help learners succeed because practice isn't conducive to learning if the learner doesn't get the problem correct (Bransford, Brown, & Cocking, 2000). My approach to this problem was to provide intra-problem adaptation. This means that if the learner was struggling to solve the current problem it could dynamically be made easier by disabling distractors, providing indentation, and combining blocks.

5.1 Research Questions

- **RQ2:** Will learners understand the intra-problem adaptation process?
- **RQ3:** What is the effect on correct completion and preference from solving 1) intra-problem adaptive Parsons problems versus 2) non-adaptive Parsons problems?
- **RQ4:** Will learners perceive that solving Parsons problems with distractors helped them learn to fix code with similar errors and write similar code?

5.2 Study Goals

My intra-problem adaptation process was based on prior research results about what makes Parsons problems easier. However, this didn't guarantee that learners would understand what was happening during the intra-problem adaptation process. Would the learners realize that disabling a distractor meant that they needed to replace the distractor with the correct block? Would the learner be able to perceive the difference between a distractor and correct block, such as incorrect case in the distractor? Would they be able to use the implicit hint of the structure of the solution if indentation was provided? Would they be able to solve the Parsons problem if the number of blocks was reduced to just three blocks? This observational study allowed us to get detailed information on the teachers' understanding of the adaptation process.

It also allowed us to ask the teachers if they preferred adaptive or non-adaptive Parsons problems and if they felt that solving Parsons problems helped them learn to fix and write code. Motivation affects the time that learners are willing to spend to learn something (Bransford et al., 2000). If learners perceive that solving adaptive Parsons problems helps them learn to fix and write code they will be more likely to stay engaged while solving them.

My hypotheses were:

- **H2A:** Most learners will understand the intra-problem adaptation process.
- **H3A:** A greater percentage of learners will correctly complete intra-problem adaptive Parsons problems than will complete non-adaptive Parsons problems.

- **H3B:** Most learners will prefer adaptive Parsons problems to non-adaptive Parsons problems.
- **H4A:** Most learners will perceive that solving Parsons problems with distractors helped them learn to fix code with similar errors and write similar code.

5.3 Intra-problem Adaptation

Parsons and Haden wrote that Parsons problems should be solvable in at most three attempts (Parsons & Haden, 2006). They didn't explain this assertion, but their goal was likely to prevent user frustration. Our log file analysis of thousands of students attempting to solve Parsons problems in an ebook, showed that some problems required up to nine attempts for 75% of the students to solve the problem (B. J. Ericson et al., 2015). On some of the easier problems, 50% of the students who never solved the problem gave up after just two attempts. On some of the more difficult problems, 50% of the students who eventually quit, gave up after six to nine attempts (B. J. Ericson et al., 2015). To help learners who are struggling on the current problem succeed, we wanted the current problem to dynamically change to be easier. I call this intra-problem adaptation.

To ensure that the user has made a genuine attempt to solve the problem, the adaptation is only available after the learner checks three distinct full solutions and still hasn't solved the problem. A full solution has at least the required number of blocks. If the user clicks the help button to initiate the adaptation before checking at least three full solutions, an alert is displayed to inform the user that help is not yet available. To draw

the user's attention when help is available, an alert appears as shown in Figure 26. This alert is only shown if the user had not yet used the help on any problems on the current page. I did not want to repeatedly show an alert each time help was available, especially if the user had already used the help, because multiple alerts can be annoying.

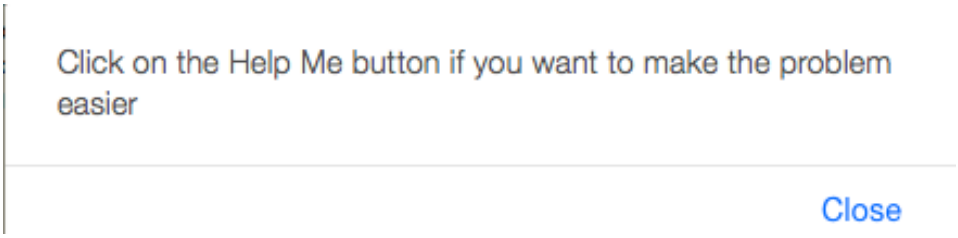


Figure 26. An alert informing the user that help is available.

When the user clicks the *Help Me* button one change is made to make the problem easier. The system has three types of changes to make the problem easier. It first disables a distractor until all the distractors have been disabled. After all the distractors have been disabled, it will provide indentation. After indentation has been provided, it will next combine two blocks into one. Successive requests for help will continue combining blocks until only three blocks are left. If the user clicks on the *Help Me* button after the solution is reduced to only three blocks, an alert appears informing the user that only three blocks are left so the user should be able to place the blocks in order.

Whenever the user clicks the *Help Me* button, an alert is first shown to notify the user of the type of change that is about to be made as shown in Figure 27. The change does not occur until the user clicks the *Close* on the alert window. This ensures that the user has been informed about what is about to happen, which should make it easier for the user to understand and track the change.

Figure 27. An alert that explains that a distractor is about to be removed (disabled).

The first type of change to make the problem easier is to disable one distractor. If the distractor was used in the solution, it moves slowly from the solution area on the right to the source area on the left as shown in Figure 28. The block is then disabled, meaning that it is grayed out over time and will not respond to attempts to move it. We are using animation here to help the learner understand what has changed. Animation is useful for grabbing attending and conveying a change over time (Baecker & Small, 1990; Chevalier, Riche, Plaisant, Chalbi, & Hurter, 2016).

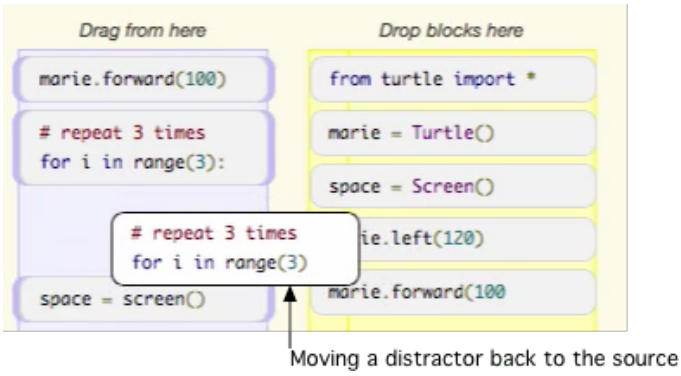


Figure 28. A distractor moving back to the source area on the left from the solution area on the right.

If a distractor was originally shown paired with the correct code, and that correct code block is still in the source area on the left, then the distractor is paired again with the correct code as shown in Figure 29. This should imply that the correct block should be used rather than the disabled distractor.

Figure 29. A distractor shown paired with the correct code and disabled (grayed out).

If there aren't any distractors in the solution it will disable a distractor in the source area on the left. This means that the block will be grayed out over time and will no longer respond to attempts to move it. Figure 30 shows all distractors in the source area disabled.

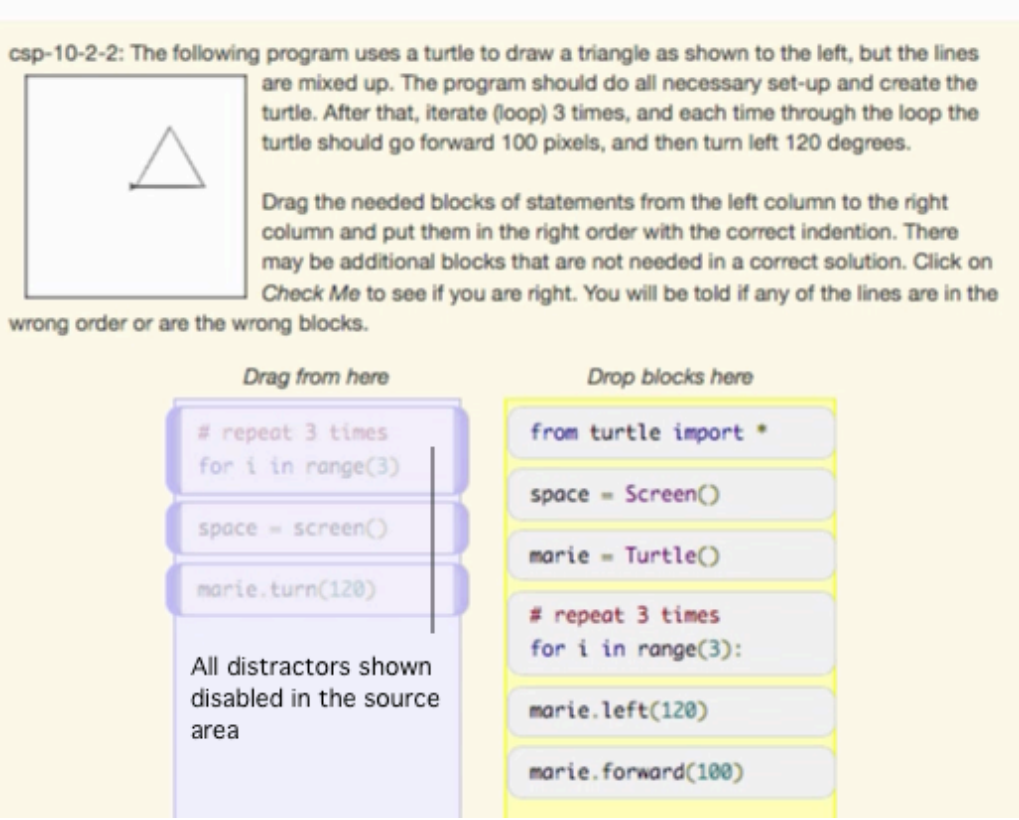


Figure 30. All distractors disabled (grayed out) in the source area.

If all of the distractors have been disabled, and the user still hasn't solved the problem, then then next type of change is to provide the indentation as shown in Figure 31. Again, we use animation to make clear to the learner what is being changed. Space is slowly added before the text in the blocks that need indentation. After indentation is provided the blocks can no longer be indented by the user, so the vertical guidelines that indicate that the user can indent the blocks are removed.

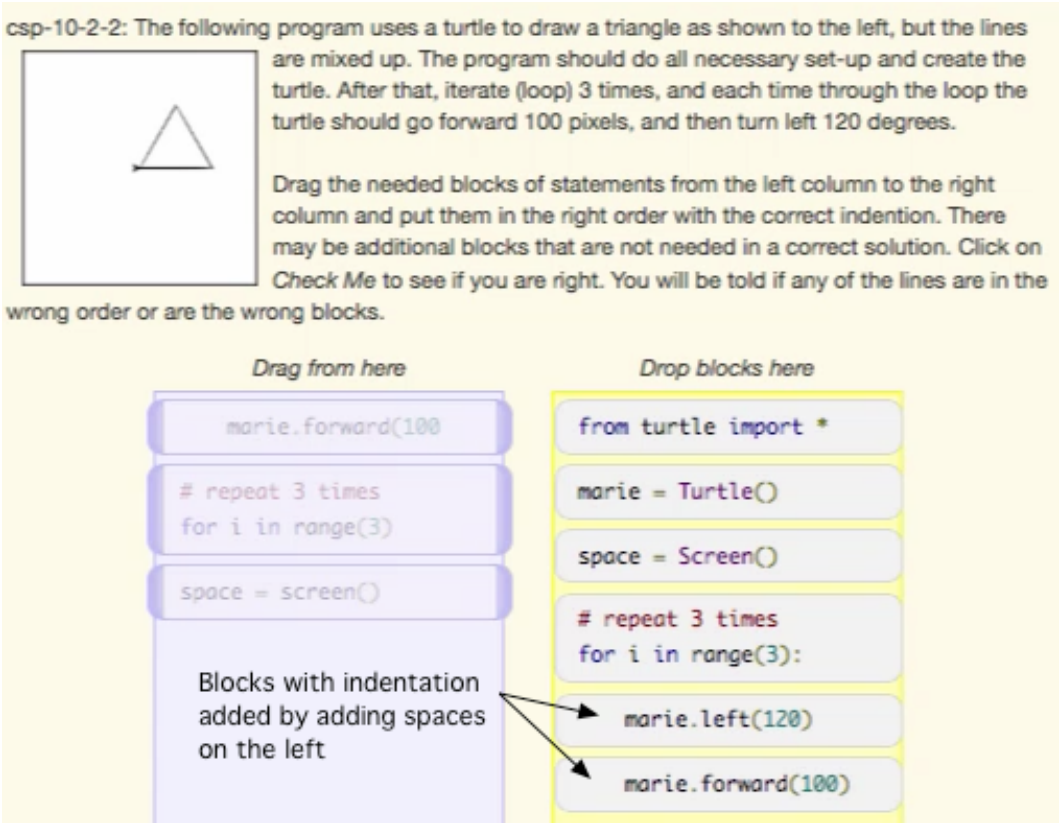


Figure 31. A Parsons problem with blocks that have had the indentation provided by adding spaces before the text.

If the user still hasn't solved the problem after the indentation has been provided then the next change to make a Parsons problem easier will combine two blocks into one. The block to be added to another is animated moving below the first block as shown in Figure 32.

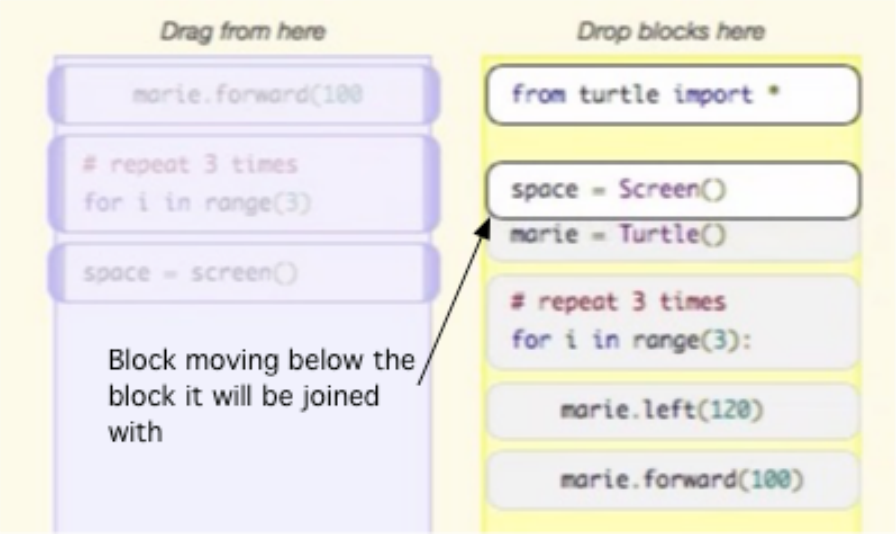


Figure 32. Moving one block below another before combining them.

The blocks are then redrawn as one block as shown in Figure 33.

Figure 33. Showing the two blocks redrawn as one after they were combined.

The software will combine two blocks into one until there are only three blocks left. At that point if the user clicks the *Help Me* button an alert appears telling the user that there are only three blocks and the user should be able to put them in the correct order. With three blocks, there are only six possible combinations. Since the user nearly

always has the first block of the solution in the correct position, this means that there are really only two possible combinations for the user to try.

5.4 User Interface Changes

We made several changes to the original js-parsons interface based on our prior studies. In my early observations of teachers solving Parsons problems I noticed that some of the teachers didn't realize that the blocks could be indented, even though the ebook told them that the body of a for loop *had* to be indented and provided example code showing the indentation (B. J. Ericson et al., 2015). The teacher's confusion was likely due to a lack of visual signifiers (Norman, 2002) to indicate that indentation was possible in the original js-parsons software. We added vertical guidelines to provide a visual signifier that the code blocks could be indented as shown in Figure 34.

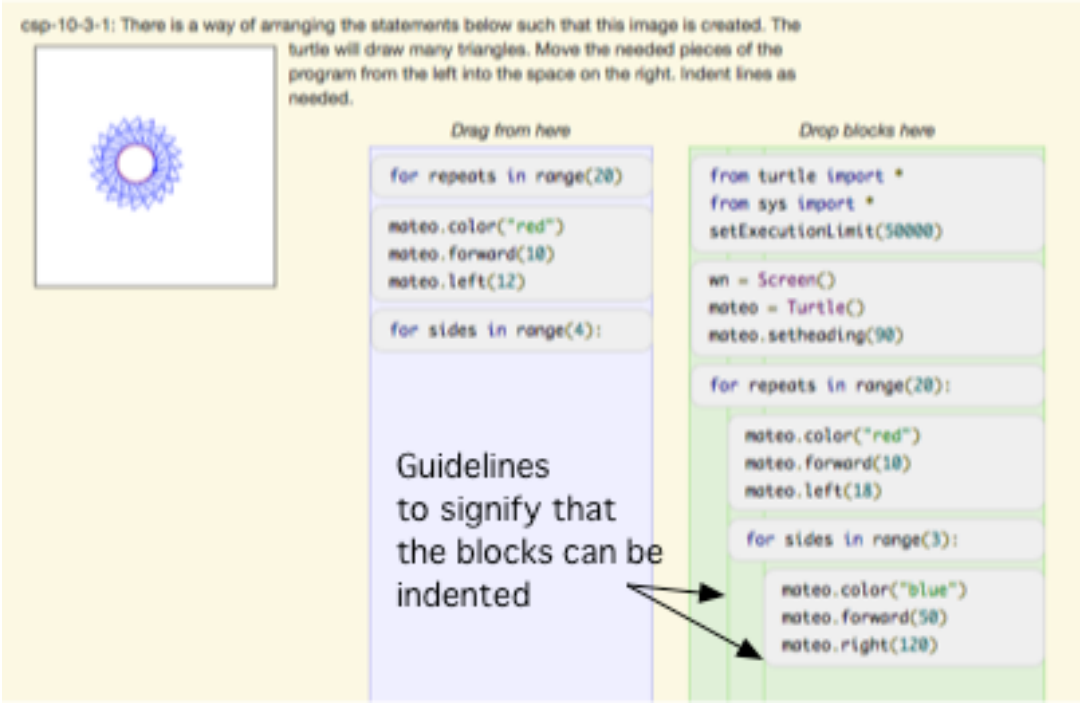


Figure 34. A solved Parsons problem with vertical guidelines to signify that indentation is possible.

Our implementation of the js-parsons software didn't check for correct indentation until all the blocks were in the correct order in the solution. Blocks that were indented incorrectly were highlighted in red on their left edge as shown in Figure 35.

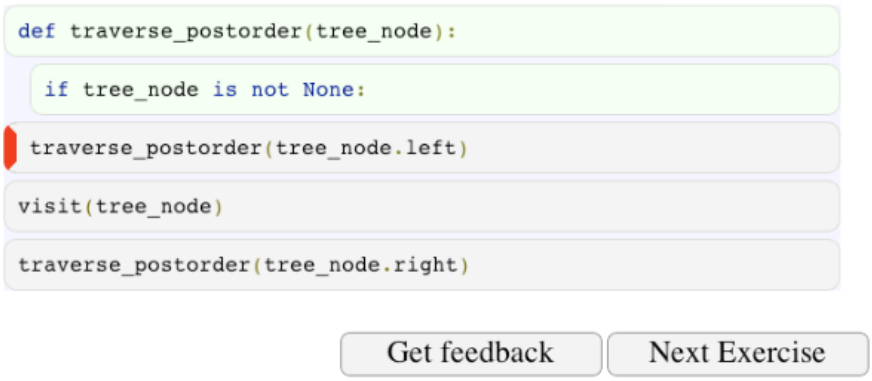


Figure 35. Block with red left edge to show that the indentation is wrong in js-parsons

In our log file analysis, we noticed that people moved blocks after being told that the highlighted blocks were not indented correctly, even though the blocks were already in the correct order. We hypothesized that users made this error because the signifier for an incorrect indentation was too similar to the signifier for out of order or incorrect blocks, which also highlighted blocks in red. We modified the software to use yellow instead of red and added arrows to indicate the direction to move the block as shown in Figure 36.

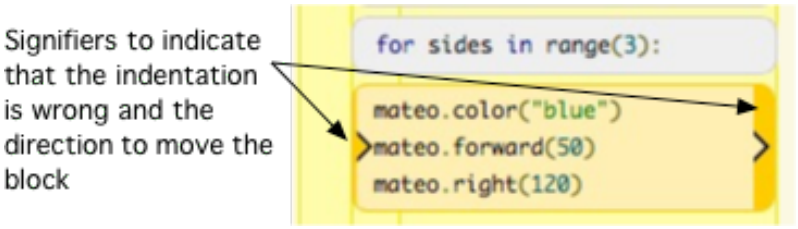


Figure 36. Signifiers to indicate that the indentation is wrong and the direction to move the block.

Another user interface design issue was how to visually indicate that a distractor was shown paired with the correct code. We did not want to leave vertical white space above and below the pairs as shown in Figure 37, which Denny, Luxton-Reilly, and

Simon had used in their paper-based Parsons problems (Denny et al., 2008), because it would limit the number of blocks that could be shown at the same time on the screen.

```
    return result;
    return word;

    String result = "";
    String result;

    if (word.charAt(i) == 'a')
    if (word.charAt(i) != 'a')

    for (int i = 0; i < word.length(); i++)
    for (int i = 0; i < word.length; i++)

    result = result + word.charAt(i);
    result = word.charAt(i);

    private String removeAllAs(String word)
    private String removeAllAs(word)
```

Figure 37. A Parsons problem with paired correct and distractor lines indicated by vertical white space. © 2008 Association for Computing Machinery, Inc. Reprinted by permission.

Originally, we displayed the correct and incorrect code blocks together, with one shown randomly either above or below the other, but without any other type of visual signifier to show that they were paired. In one of our pilot tests 20% of the undergraduate students used both the correct and distractor blocks in the same solution, which indicated that this approach was not successful at signifying to the user that the blocks were paired, and they only needed to use one in the solution. After that pilot study, we added purple edge decorators to indicate the pairing as shown in Figure 38.

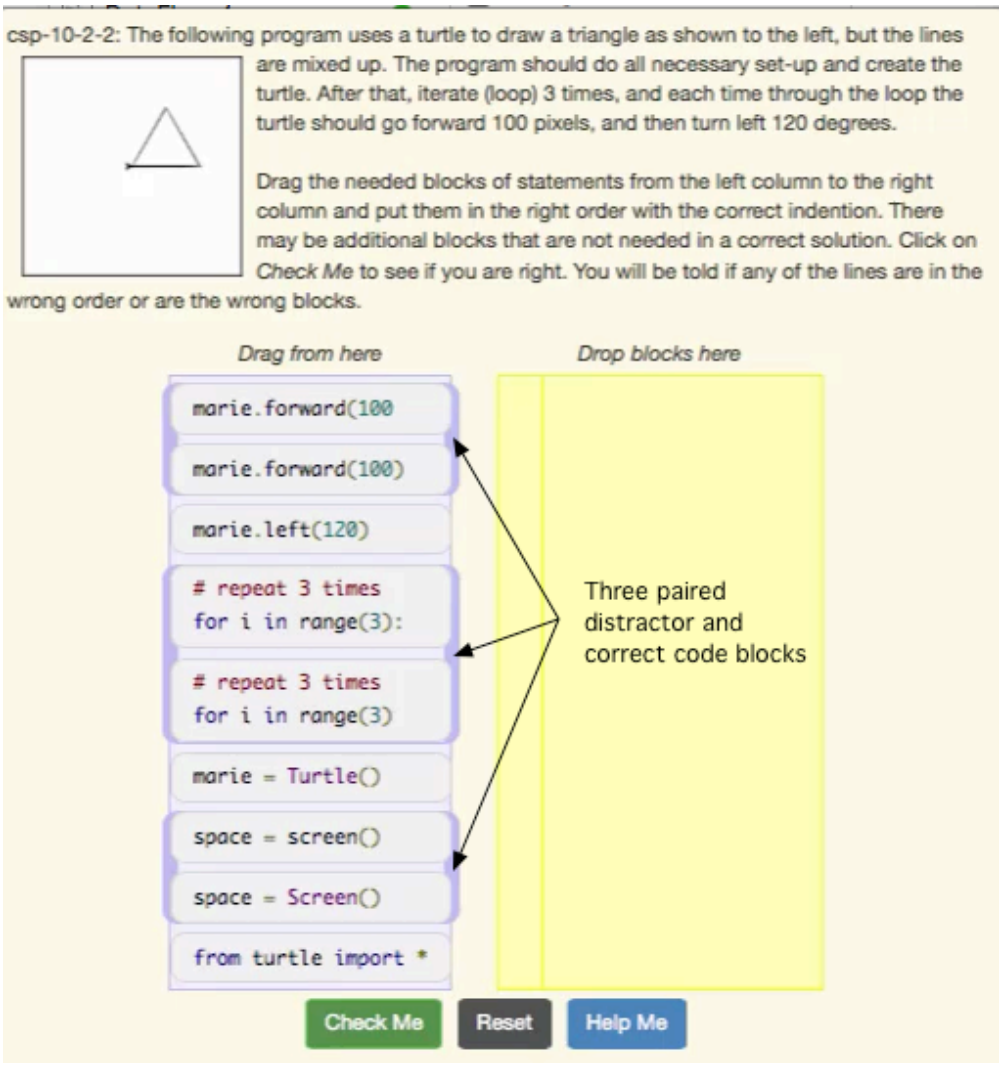


Figure 38. A Parsons problem with purple edge decorators signifying the pairing of the correct block and the distractor.

5.5 Study Materials

I created two versions of an ebook using the first four chapters of a teacher ebook for the Advanced Placement (AP) Computer Science Principles (CSP) course that our research group has been developing for several years using a worked example plus

practice approach (B. Ericson, M. Guzdial, et al., 2015). The AP CSP course is a course offered in secondary schools and is similar to a college level course in computing for non-majors. This course includes basic programming concepts such as variables, loops, conditionals, and functions. The teacher ebook was created to provide teachers with free professional development in order to increase their knowledge and confidence. The first four chapters include instruction on using variables in programs with numbers and strings. Chapter five originally contained instruction on using variables with LOGO style virtual turtles which draw as they move (Papert, 1980).

I added additional material to chapter five from the teacher AP CSP ebook on loops, lists, and the range function. I also added Parsons problems that used loops and nested loops and created several additional Parsons problems for a total of 20 Parsons problems in chapter five. Ten of the Parsons problems were adaptive (had a *Help Me* button) and ten were not.

5.6 Formative Study

To test the study materials, I observed three undergraduate students as they individually worked through the materials in chapter five. The undergraduates received five points of extra credit in an unrelated course for taking part in the study. They all had prior experience with textual programming, so they were more experienced than the intended audience. The expectation was that if the undergraduate students needed help, it would indicate that the more inexperienced teachers would also need help. However, the undergraduate students solved all of the Parsons problems without using the *Help Me* button. I dropped four of the Parsons problems after this study since the students solved

these easily. I also added distractors to the problems and changed half of the problems, to use distractors randomly mixed in with the correct code blocks, rather than paired distractors, in order to make the problems harder. I observed two more undergraduate students and found that they needed help, which suggested that the problems were now of the desired difficulty.

All five undergraduates said that they preferred to have the help available than not, and preferred to have the distractors shown paired rather than randomly mixed in with the correct code blocks. They all felt that solving Parsons problems *would* help them learn to fix and write code.

5.7 Observational Study Materials

In group A's version of the ebook the first Parsons problem was not adaptive and the second was as shown in Table 13. In group B's version of the ebook the first problem was adaptive and the second was not. After the first Parsons problem, the user solved pairs of adaptive or non-adaptive Parsons problems as shown in**Table** 13. Alternating adaptive and non-adaptive problems gives us two advantages. First, we control for bias introduced by setting an expectation with the first type of Parsons problem encountered. Some of the participants would see adaptive problems first and some would see non-adaptive first. Second, we could contrast learner performance on the same problems where some users solved the adaptive version and some the non-adaptive version.

Table 13. Which problems were adaptive in the two groups. No means not adaptive and yes means intra-problem adaptive.

Problem	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Group A	No	Yes	Yes	No	No	Yes	Yes	No
Group B	Yes	No	No	Yes	Yes	No	No	Yes
Problem	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Group A	No	Yes	Yes	No	No	Yes	Yes	No
Group B	Yes	No	No	Yes	Yes	No	No	Yes

Half of the problems had paired distractors and half had the distractors randomly mixed in with the correct blocks as shown in Table 14.

Table 14. The use of paired and un-paired distractors. Yes, means the distractors were paired and no means they were randomly mixed in with the correct code.

Problem	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	Yes	Yes	No	No	Yes	Yes	No	No
Problem	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
	Yes	Yes	No	No	Yes	Yes	No	No

Chapter five also included four problems that required the user to fix code with errors similar to those in the Parsons problems distractors and three problems that required the user to write code from scratch.

5.8 Observational Study Procedure

We conducted a within-subjects observational study of teachers with less than three months of textual programming experience as they worked through a chapter of an ebook that used a worked example plus practice approach. The practice problems were

eight adaptive and eight non-adaptive Parsons problems. The teachers worked on their own through the first four chapters and then were observed remotely for two hours using Zoom videoconferencing software (https://zoom.us) as they worked through chapter five with the adaptive and non-adaptive Parsons problems.

5.8.1 Recruitment

I sent email to my mailing list of over 500 secondary teachers and also sent email to six other instructors who lead teacher professional development to ask them to forward the email to their participants. The email stated that we were seeking teachers with less than three months of experience during the last two years with textual programing languages to try a new approach to learning Python. Teachers who completed the study earned a $75 gift card. Teachers had to complete four chapters on their own and then be observed remotely using Zoom videoconferencing software as they completed the fifth chapter. They were asked to complete the chapters by the end of August 2017. The email contained links to the consent form and demographic survey.

5.8.2 Study Procedure

Interested teachers first gave consent online. Next, they filled out a survey, which asked for demographic information such as the gender they identified with, age, race, certification/license, number of years teaching, number of years teaching computing courses, if they had less than three months of textual programming experience, and their familiarity with computing concepts such as variables, loops, conditionals, and lists. Qualified teachers were randomly assigned to group A or B. In group A, the first Parsons

problem was not adaptive and in group B it was adaptive as shown in Table 13. Qualified teachers were sent an email welcoming them to the study. The email contained the URL for their version of the ebook, their login, and their password. The teachers were instructed to work through the first four chapters on their own and then contact me to arrange for a two-hour observation using Zoom videoconferencing software. Teachers were sent an email every week, which reminded them that they could quit the study at any time and asked that they contact the first author to arrange an observation when they had completed chapter four. The procedure is shown in Figure 39.

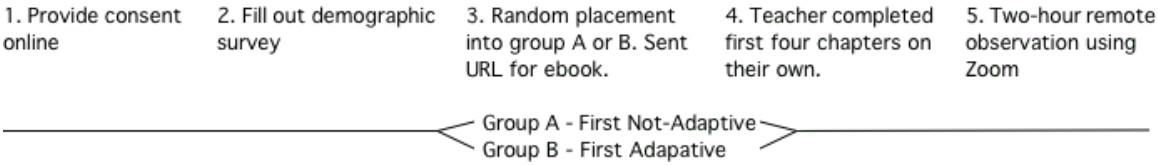


Figure 39. Procedure for the teacher observation study.

Twenty-six teachers applied to be in the study. Of these some had more than three months experience in a textual programming language and were disqualified. Eighteen teachers enrolled in the study, but five withdrew from the study because they didn't think that they had time to finish before the deadline, and another two teachers didn't complete in time. This chapter is reporting on the 11 teachers (5 in group A that solved a non-adaptive Parsons problem first and 6 in group B that solved an adaptive Parsons problem first) that were observed working through chapter five. While 11 is a smaller sample size than desired for statistical tests, it still can yield a great deal of qualitative data and provides the ability to check understanding and preferences.

Six (55%) of the teachers identified as female and five (45%) as male as shown in Table 15. Nine teachers (82%) had not taught computing courses for more than two

years, but one teacher had 10 years of experience and the other had 15 years. These experienced computing teachers were business or instructional technology teachers who often teach computer application courses and/or keyboarding. The teachers came from seven different states in the United States: Alabama, California, Colorado, Illinois, Georgia, Massachusetts, and New Jersey. Eight (73%) of the teachers had prior experience in Scratch (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010), three (27%) in App Inventor (Honig, 2013), three (27%) in Java, two (18%) in Alice (Pausch, 2008), two (18%) in JavaScript, one (9%) in Squeak, one (9%) in Snap, and one (9%) in Python. Eight (73%) teachers reported that they had less than one year of experience programming in a drag and drop environment.

Table 15. Teacher Demographics and Number of Problems Used Help On

Id	Group	Gender	Race	Age	License / Certification	Years teaching comp.	Number problems used help on
T1	B	Female	White	40	Business	10	4
T2	B	Female	White	26	Biology	0	0
T3	A	Female	White	30	Eng, Math, & Science	1	0
T6	A	Female	Hispanic	55	Math	1	4
T8	A	Male	Arab	37	Bio. & Chem.	2	0
T9	B	Male	White	33	History, English, & Tech Specialist	2	4
T10	A	Male	White	42	Instructional Tech.	15	1
T11	A	Male	White	42	Language Arts – Adding CS	0	1
T13	B	Male	White	43	Didn't answer	1	0
T14	B	Female	Asian	26	Math	1	1
T18	B	Female	White	59	Math – Alt.	1	1

5.9 Use of the Intra-Problem Adaptation

Of the 11 teachers, seven (63%) used the intra-problem adaptation by clicking the *Help Me* button at least once during the two-hour observation as shown in Table 15. Three teachers (T1, T6, and T9), used the adaptation on four problems. Four teachers used the adaptation on one problem (T10, T11, T14, and T18).

One might think that there would be less total attempts on the adaptive problems than on the non-adaptive. Eight (72%) of the teachers had a lower total number of attempts on the adaptive versus the non-adaptive problems, while three teachers had a higher number of attempts for the adaptive versus the non-adaptive problems as shown in Table 16. Note that teacher 9 had a much higher total on adaptive problems (38) than on the non-adaptive problems (21).

Table 16. The number of problems where the teacher used the adaptation, the total number of attempts for both the adaptive and non-adaptive problems, and the difference.

Teacher Id	Number problems used adaptation	Total attempts on adaptive problems	Total attempts on non-adaptive problems	Non-adaptive total minus the adaptive total
T1	4	35	49	14
T2	0	18	15	-3
T3	0	13	20	7
T6	4	37	39	2
T8	0	9	14	5
T9	4	38	21	-17
T10	1	21	22	1
T11	1	18	12	-6
T13	0	15	22	7
T14	1	16	19	3
T18	1	17	21	4

The intra-problem adaptation was used a total of 16 times on nine (50%) of the 18 problems. It was available on half (eight) of the problems, so the usage rate was 18% of the 88 (8 * 11) possible times. Table 17 shows the problem, the number of blocks and the number of distractors originally in the solution, and the teachers who solved each problem after distractors were disabled, indentation was provided, or blocks were combined. Four times the teachers were able to solve the problem after just one distractor block was removed (T1 and T9 on problem 1, T6 on problem 6, and T18 on problem 13). One time a problem was solved after all four distractors were removed (T1 on problem 9). Two problems were solved after all the distractor blocks were disabled and the indentation was provided (T6 on problem 11 and T1 on problem 13). Nine times a problem was solved after all distractors were disabled, the indentation was provided, and blocks were combined. Four teachers were able to solve the problem after one block was combined with another (T9 on problem 4, T6 on problem 7, and T9 and T14 on problem13). T11 reduced problem 11 to three blocks before solving it. T10 also reduced problem 11 to three blocks, but then hit reset and started over and then solved it without help.

Two teachers gave up and didn't solve two of the non-adaptive problems. Teacher T9 gave up on problem 11 after 11 attempts saying, *"I don't know what I am doing on this one."* He had two blocks swapped in his solution. Teacher T6 gave up on problem 16 after 15 attempts to solve it and asked me for help. She was missing one of the for loops in a nested for loop.

Table 17. The Parsons problems where intra-problem adaptation was used.

Problem	Number of Blocks in solution (# distractors)	Solved after distractors were removed - # removed	Solved after indentation was provided	Solved after blocks were combined - # combined
1 – Draw L	7 (4)	T1-1, T9-1		
4 – Draw A	7 (4)			T9-1
6 –Draw N	7 (4)	T6-1		
7 –Draw F	7 (4)			T6-1
9 – Draw Rectangle	6 (4)	T1-4		
11 –Draw Spiro	6 (3)		T6	T10-3, T11-3
13 – Stamp Line	7 (4)	T18-1	T1	T9-1, T14-1
15 – Stamp X	9 (4)			T6-2
16 – Stamp Squares	9 (4)			T1-3, T9-4

5.10 A Closer Look at the Last Six Problems

The intra-problem adaptation was used 10 times on the last six problems compared to six times on the first 10 problems, so it might be useful to examine the last six problems in more detail. One way to compare the adaptive and non-adaptive problems is to look at the average number of attempts for the two groups on the same problem. In problems 11, 14, 15, and 16 the adaptive group had a lower average number of attempts than the non-adaptive group, but in problems 12 and 13 the non-adaptive group had a lower average number of attempts than the adaptive group.

Problems 11 and 12 were very similar as you can see from Figure 40, so it is interesting to see that while some teachers clearly struggled to solve problem 11 as shown in Table 18, none of the teachers took more than 3 attempts to solve problem 12. This indicates near transfer, which is being able to apply what you learned in one problem to another similar problem.

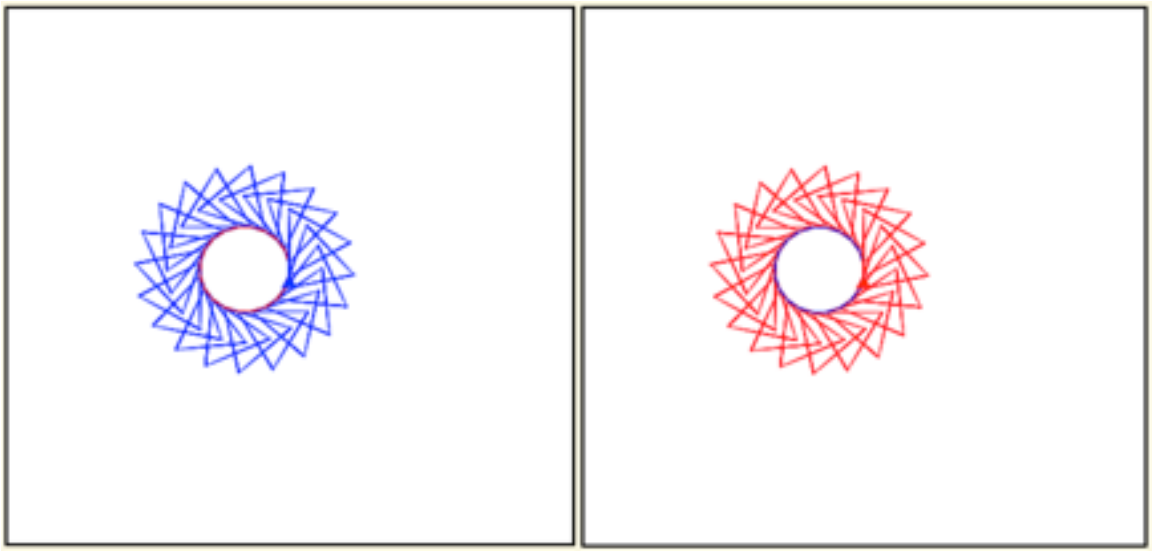


Figure 40. What the turtle draws on Problems 11 (left) and 12 (right)

Table 18. Number of attempts per teacher on each of the last six problems. The A following the problem number indicates adaptive and the NA not adaptive. The Y indicates the problems where the help (adaptation) was used. The * indicates that the teacher did not solve the problem.

Group A	**11 – A**	**12 - NA**	**13 - NA**	**14 - A**	**15 - A**	**16 - NA**
T3	2	1	8	1	2	4
T6	6 - Y	2	10	2	8 - Y	15*
T8	2	1	2	1	1	4
T10	8 - Y	3	2	1	3	7
T11	7 - Y	1	4	1	1	1
Average (Std Dev)	*5 (2.8)*	*1.6 (0.9)*	*5.2 (3.6)*	*1.2 (0.4)*	*3 (2.9)*	*6.2 (5.4)*
Group B	**11 - NA**	**12 - A**	**13 - A**	**14 - NA**	**15 – NA**	**16 - A**
T1	10	3	7 - Y	13	14	8 - Y
T2	2	2	3	1	1	3
T9	11*	2	9 - Y	1	2	12 - Y
T13	5	2	2	8	1	1
T14	3	1	7 - Y	1	10	2
T18	3	2	6 - Y	4	3	1
Average (Std Dev)	*5.7 (3.9)*	*2 (0.6)*	*5.7 (2.7)*	*4.7 (4.9)*	*5.2 (5.5)*	*4.5 (4.5)*

5.11 An In-Depth Look at Problem 13

To better understand the effectiveness of the adaptation process it is helpful to take an in depth look at one problem. Four teachers used the intra problem adaptation on problem 13, which was the largest number of teachers who used the adaptation on any problem. Problem 13 was also the only problem that was solved after each level of

adaptation: after distractors were disabled, after the indentation was provided, and after blocks were combined.

Problem 13 asked the learner to stamp three turtle shapes in a line as shown in Figure 41. This problem had four paired distractor blocks. The distractor blocks had common syntax errors like the wrong case, missing parentheses, missing quotes for the string “turtle”, and a missing colon at the end of the for statement.

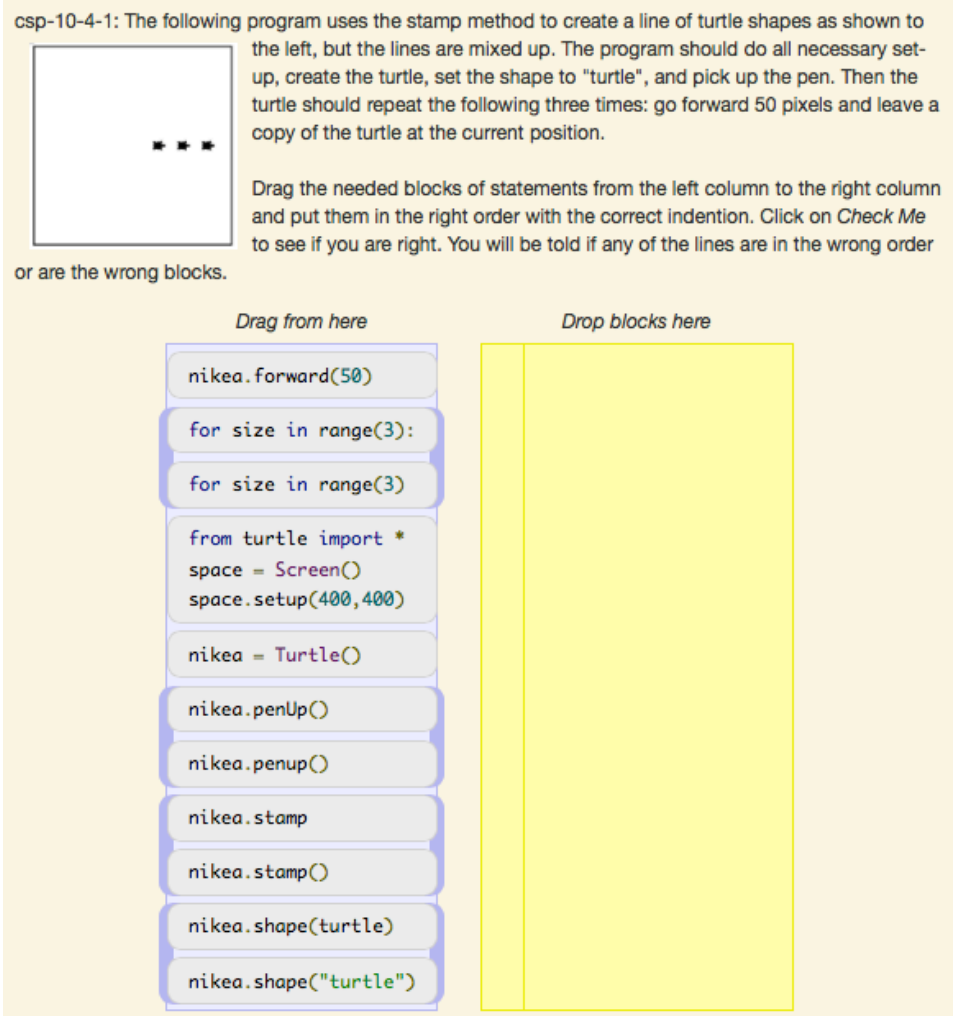


Figure 41. Problem 13, Stamp three turtle shapes in a line.

5.11.1 Solving the problem after a distractor was disabled

Teacher T18 solved this problem after one distractor was removed. When she had the solution as shown in Figure 42 she said, *"I don't have my other program to look at to see if I have to put a shape turtle in there. I am just going to run this and see what happens."* This is interesting because it indicates that she didn't realize that the correct and incorrect code were shown paired with the purple edges indicating the pairs, since that should have been a clue to her that she needed to use one of the two shape blocks.

The solution check told her the solution was too short and that she needed to add more blocks. She said, *"I think I must need the shape in there somewhere"*. She scrolled back up and read that she could change the shape using the shape function.

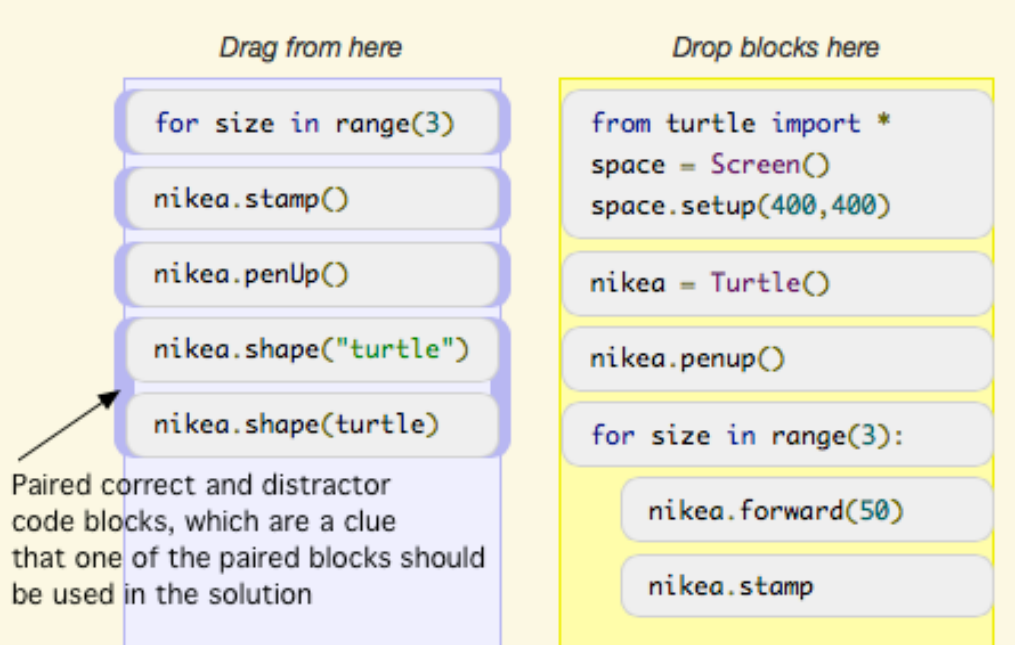


Figure 42. First solution which did not set the turtle shape.

She then added the correct shape block. The solution check highlighted the stamp block. She tried moving it before the forward instead, but that was still wrong. She said, *"I don't have the other program to refer to. So, I don't know what I am doing wrong."* She actually had example code above this problem, but kept scrolling below the problem instead of up to see the example. I told her to hit the check-me one more time. The alert said that help was available. She clicked on the *Help Me* button and an alert said that it would remove (disable) an incorrect block. She clicked on *Close* and the stamp moved back to the source and paired with the correct block and then grayed out. She said, *"Ah, okay I need the one with the thing (parentheses). Oh gosh, I should have noticed that.*

Oh my gosh how am I going to help students if I didn't even notice that that (motioned to the block) was missing that (motioned to the parentheses).'' It is well known that experts pick up on details like missing parentheses that novices do not (Bransford et al., 2000). Solving Parsons problems with paired distractors with syntax errors may help novices learn to spot these types of errors.

5.11.2 Solving the problem after the indentation was provided

Teacher T1 solved this problem after indentation was provided. The first time she checked her solution it was too short. She had forgotten to create the turtle. She added the block to create the turtle and then checked her solution which highlighted the last two blocks in the loop as needing to be moved or replaced. She then moved the last block, the *forward*, before the *penup* and checked her solution again. It highlighted the last two blocks again as shown in Figure 43.

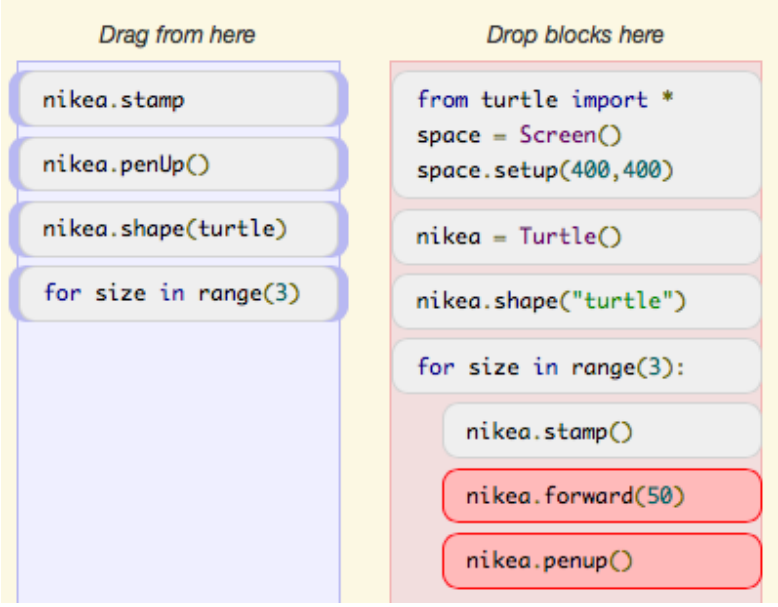


Figure 43. After adding the block to create the turtle and moving the last block up one.

Next, she added the distractor that set the shape to turtle after the last block in the loop, but removed it again without checking the solution. She then clicked on the help button, but the alert was displayed saying that she needed to make three full attempts before help would be provided. The first check didn't count since it wasn't a full solution. She then moved the *penup* block before the forward again. This time when she checked her solution an alert appeared to notify her that help was available. She clicked on the help button four times to disable the four distractors, but she hadn't used any of them in her solution. Then she clicked on the help button again and the alert said that indentation would be provided. She clicked on the *Close* button and space was slowly added before the text in the blocks to provide the indentation as shown in Figure 44. Notice that the guidelines were removed, since the user no longer needed to provide the

indentation. Also, notice that the *penup* block shouldn't be in the loop since it isn't indented.

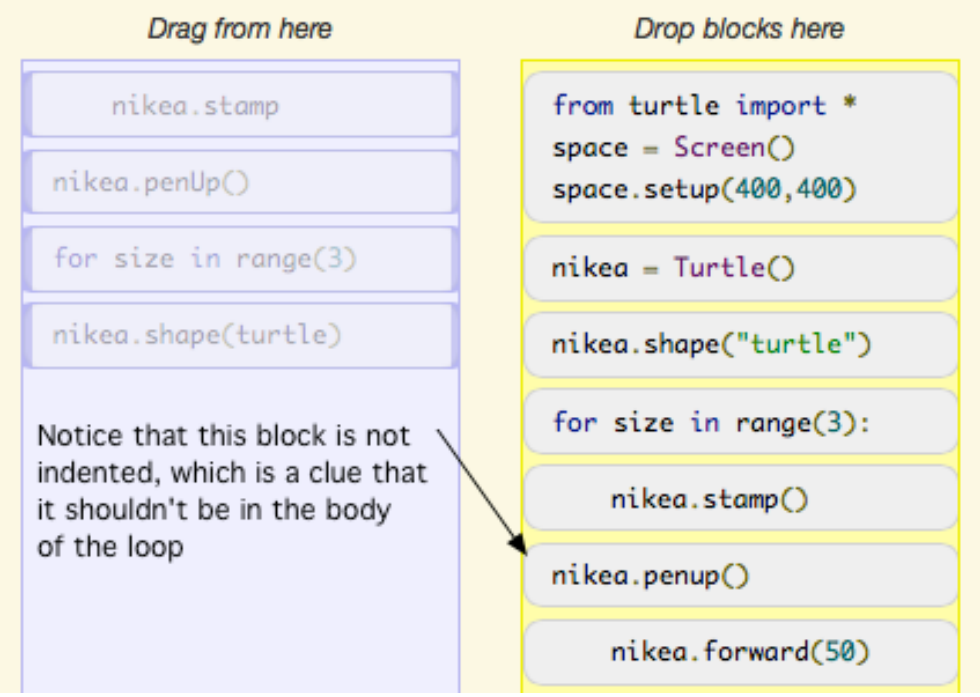


Figure 44. After all distractors were disabled and indentation was provided.

After the indentation had been provide she picked up the stamp block and tried to indent it, but could not. She then asked me, *"So these last three (blocks) aren't supposed to be indented?"* I answered, *"Not the penup."* She then said, *"But, these two are?"* and moved her cursor over the stamp and forward blocks. I answered, *"So, what it did was provide the indentation."* She then moved the *penup* block before the loop and checked her solution. The last block was highlighted as needing to be moved. She then moved the forward before the stamp and that solution was correct. She hadn't realized that she should pick up the pen using the *penup* block before the turtle moved forward or it would draw a line when the turtle moved, even though the example showed the *penup* before the

loop. While it really wouldn't matter in practice, there is no need to put the *penup* in the loop, because once it is picked up it will stay up until it is put down again.

5.11.3 Solving the problem after combining blocks

Teacher T9 originally dragged the *penUp* (wrong case) distractor into his solution, but then immediately moved it back to the source and dragged out the correct *penup* block instead. This indicates that the pairing helped him focus on what was different about the two blocks. He then checked a solution that didn't include the block to set the turtle shape, and was told that the solution was too short. He, like teacher T18, did not seem to realize that the distractor and correct blocks were being shown paired, which meant that he should use at least one of the two blocks in the pair to set the shape.

After being told that the solution was too short, he dragged in the distractor block for setting the shape. When he checked the solution, it highlighted the distractor and the two blocks in the loop that had to be moved for a correct solution as shown in Figure 45. Like teacher T1 he also had included the *penup* block in the loop, even though the previous example shows it before the loop.

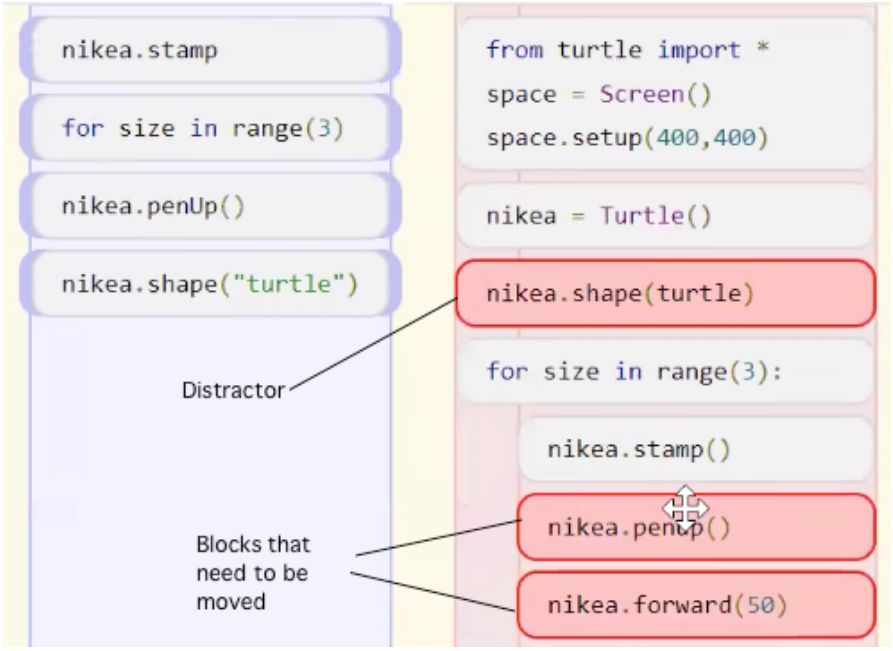


Figure 45. This highlights the distractor that has been used in the solution and the two blocks that are out of place.

He tried moving the *penup* block back to the source on the left, but the solution check said that the solution was too short again. He then moved the *penup* block back into the solution as the last block in the loop. The solution check highlighted the same blocks as before. He then clicked on the *Help Me* button, but it told him that he needed to make three full distinct attempts before he could get help. He moved the forward block to be the first block in the loop. When he checked the solution this time the alert said that help was now available. He clicked the *Help Me* button and it moved the shape distractor that he had used back to the source and paired it again with the correct block and grayed out the distractor. He replaced the distractor with the correct block and checked the solution again. It highlighted the *penup* block as still being out of place. He then clicked the *Help Me* button twice, which disabled two of the unused distractors. He tried to move the

penup to be the first block in the loop, but the solution check still highlighted it as needing to be moved or replaced. He clicked the *Help Me* button to disable the last unused distractor. He then clicked the *Help Me* button which provided the indentation. The solution now looked as shown in Figure 46.

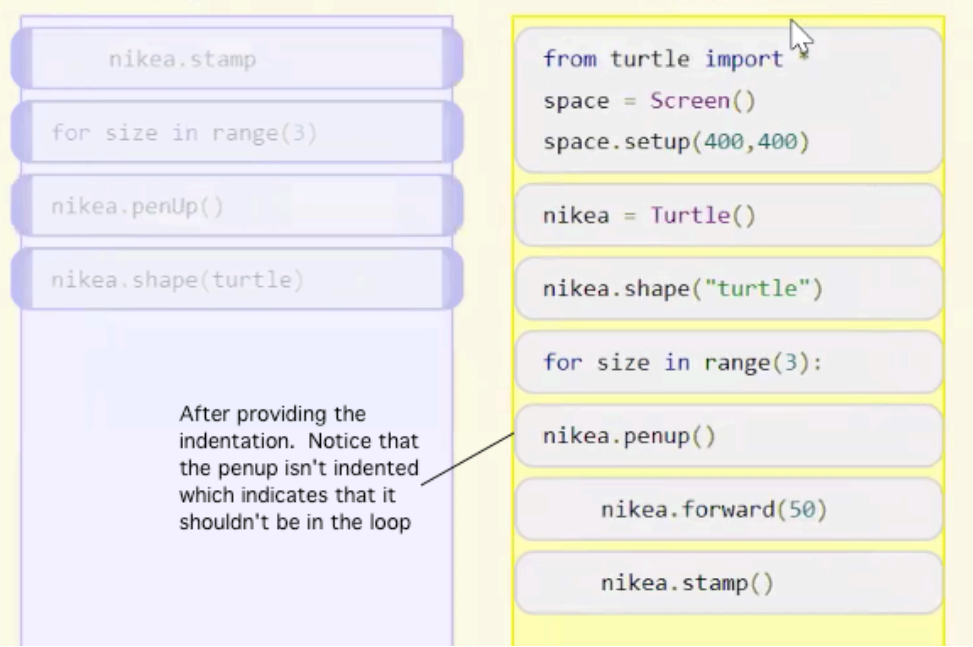


Figure 46. After all the distractors have been disabled and the indentation has been provided.

He started to move the *penup* block, but stopped and clicked the *Help Me* button again, which moved the *penup* block below the shape block and then combined them as shown in Figure 47.

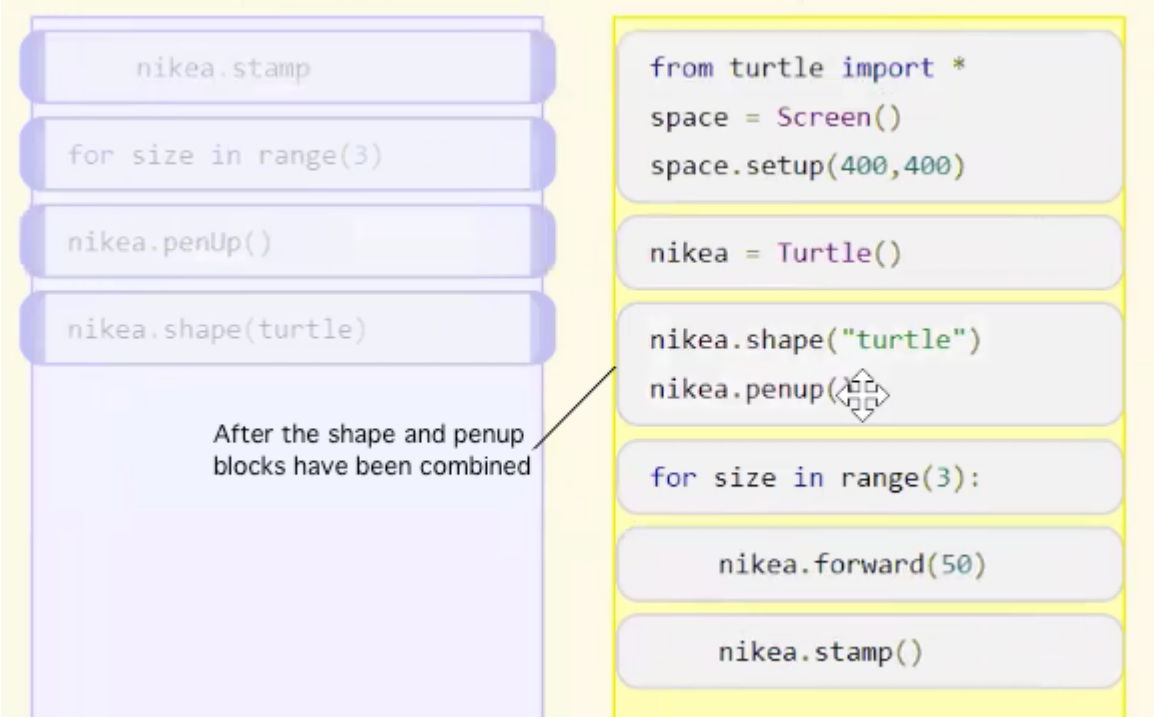


Figure 47. After the penup block was combined with the shape block outside the loop.

This example indicates that the paired distractors did help the teachers focus on the difference between the distractor and correct block, but the teachers didn't always realize that they had to use one of the two blocks. It also shows that providing indentation didn't always help the learner. It should have indicated to the learner that the *penup* block shouldn't have been in the loop, since it wasn't indented. This teacher didn't understand that implicit hint since he didn't get this problem correct until the *penup* block was combined with the shape block.

5.12 Learning to spot common syntax errors

The Parsons problems contained distractors with common syntax errors like the wrong case on function names and a for loop with a missing colon at the end. In Python

a for loop must have a ':' at the end of it. Teacher T10 was solving problem 10 as shown in Figure 48 which had paired distractors when he asked, *"Why are there two of the same?"* He didn't notice that the first block was missing the colon at the end of the statement, while the second block had it. He dragged the distractor into his solution. When he checked the solution, the distractor block was highlighted as needing to be moved or replaced. He then replaced it with the correct block. I then asked, *"Do you see what is different between the two?"* He responded, *"No, I don't"* and started to read the block out loud. I explained that the correct block had the colon at the end and the colon was required. He said, *"Oh. Oh. All right"* and chuckled.

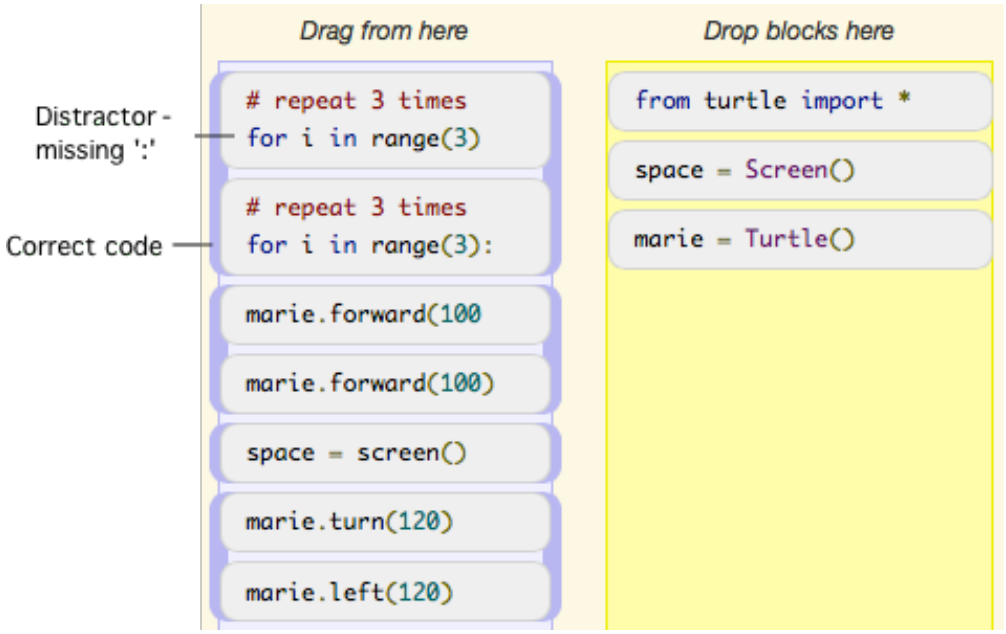


Figure 48. Paired distractor and correct code blocks for problem 10. Note that the distractor is missing the colon at the end of the statement.

The next two Parsons problems had distractors randomly mixed in with the correct code and both of those problems had a distractor with a missing colon, but he didn't use those distractors in his solutions. Next, he solved the first fix code problem shown in Figure

49. He tried to compile the code which printed that there was a syntax error of bad input on line 9. He scrolled up to look at his previous correct Parsons problem solution and then down again to look at the fix code problem and added the colon at the end of line 9. This demonstrates that he was able now to spot the missing colon at the end of the for loop, but he needed to compare his line of code with the correct line to spot the difference.

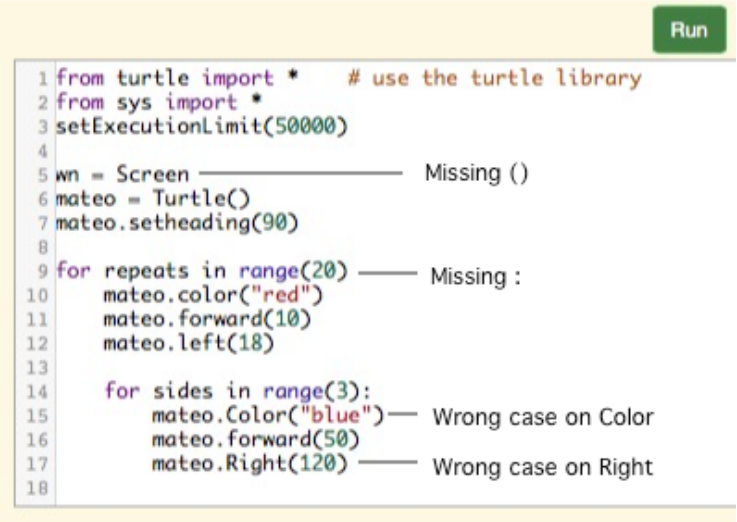


Figure 49. First fix code problem with four errors

5.13 Problems with the Adaptation Process

The observations provide evidence that some of the teachers found parts of the adaptation process confusing. In particular teachers were confused when distractors were disabled that they hadn't used in their solution. The distractors would highlight and then gray out over time, but the teacher's attention was on the solution area on the right and not on the source area on the left. Teacher T10 said, *"So I clicked it (the Help Me button) several times to see it change in the right column (the solution), and I didn't see it. I didn't realize what it was doing to the left column (the source)."*

Teacher T10 also struggled to solve problem 11. He first used the help to disable all four distractors, but he hadn't used any of them in his solution. The next time he clicked the help button it provided indentation as shown in Figure 50. The indentation should have indicated to him that two of the blocks were in the wrong place since the indentation didn't match their location, but he didn't move those blocks.

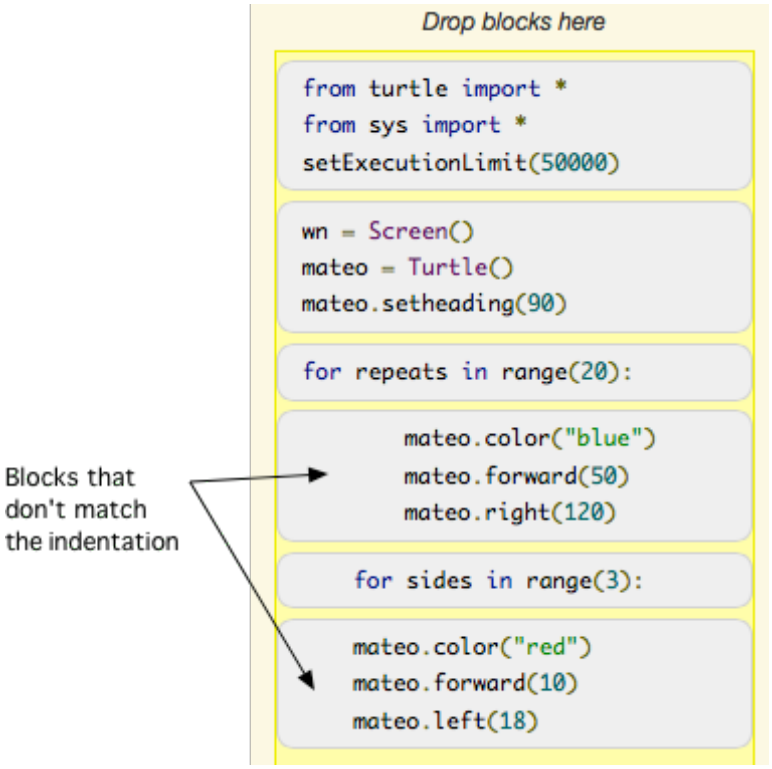


Figure 50. Teacher T10's solution after indentation was provided

He used the *Help Me* to combine blocks until the solution was down to only three blocks as shown in Figure 51, but the blocks were still not in order. The next time that he clicked the help button the alert said that there were only three blocks left and he should be able to put them in order, but he just closed the alert and didn't seem to read it. Then he reset the problem to start over, saying *"I'm lost"*. However, he then did manage to

solve the problem without using the help again. When we discussed the adaptation after he had finished the chapter he said, *"So if I proceed all the way through, it's gonna give me the whole solution, right*?". This indicates that he didn't realize that he could only use the Help Me until he had three blocks left and that he would still have to put those blocks in the correct order.

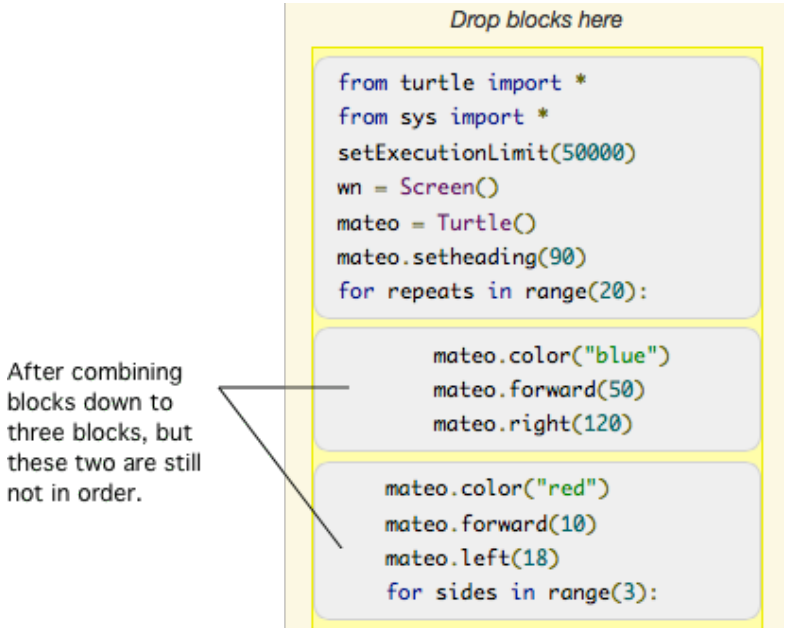


Figure 51. Teacher T10's solution after combining down to three blocks

5.14 Preferences and Perceptions

If the teacher did not use the intra-problem adaptation (help button) at all during the observation, I demonstrated it to him or her after the chapter was completed. I then asked the following questions.

1) Some of the mixed-up code (Parsons) problems had a *Help Me* button (allowed intra-problem adaptation) and some did not. Which did you prefer and why?

2) Some of the mixed-up code (Parsons) problems had purple edge decorations to show the pairs of correct and incorrect code (distractor) blocks and some had the incorrect blocks randomly mixed in with the correct blocks. Which did you prefer and why?
3) Do you feel that solving the mixed-up code problems with distractors helped you when you had to fix code with errors or when you had to write code?

5.14.1 Preference for Intra-Problem Adaptive or Not Adaptive

Nearly all of the teachers said that they preferred the adaptive Parsons problems where help was available versus the non-adaptive. One teacher even asked, "*Why isn't it letting me do the Help Me?"* when she was struggling to solve a problem that did not provide help. However, some of the teachers were concerned that they didn't want the help to become a *"crutch"* and were worried that some students might abuse the help. One teacher said, *"I like the 'Help Me' button and I like that it comes on after 3 tries. The kids, I don't want them to be able to, you know, right away go to the Help Me. They need to try it first"*. She also said, *"I noticed with me, when I got all those wrong answers and I went to the next one, I was able to do it."* This matches what we see in the data. Teachers who struggled on one problem and used the help, often solved the next problem in less attempts without any help. This provides evidence of near transfer which means that the learners are able to transfer what they have learned on one problem to a similar problem.

One teacher thought that the help was good for formative assessments (those used to check student understanding and modify instruction), but didn't want it available on

summative assessments (those used for grades). In our software, the adaptation is optional and is off by default. One teacher thought that it might be better to show a similar problem or provide audio help rather than dynamically change the difficulty of the problem.

5.14.2 Preference for Visually Paired or Not

Many of the teachers had not noticed the purple edge decorations that were intended to visually signify that the correct block and distractor block were being shown paired and that they should use one of the two blocks. One teacher said, *"Honestly, were those there when I was doing that problem?"* Once all the teachers were aware of the edge decorators, all but one teacher thought that paired distractors were easier than un-paired. One teacher said he noticed the edge decorators, but didn't think they made much of a difference. One teacher preferred the un-paired distractors, because she wanted more of a challenge. In our inter-problem adaptation that we added after this study, if the user has solved the current problem too quickly then the next problem can be made harder by adding more distractors or by randomly mixing in the distractors with the correct code rather than pairing each distractor with the correct code block. The goal is to keep the learner challenged, but not frustrated.

5.14.3 Perception of the Usefulness of Parsons Problems

All of the teachers said that they felt that solving Parsons problems with distractors helped them when they had to fix code with the same type of errors as the distractors. One teacher said, *"Yes, it kept reminding me. Hey, that requires a colon, that*

requires parentheses, that requires lower case or upper case. That is what I was struggling with in the beginning." All of the teachers said that they felt that solving Parsons problems with distractors also helped them when they had to write code from scratch. Teacher T18 said, "*Yes, 'cause you realize how, what's important, that you do need to focus on the uppercase, lowercase, making sure you've got the parentheses afterwards, and the colons, that those are all important. And sometimes really, you know, you'll be sitting there writing code, and you just won't see it."* Teacher T9 found solving Parsons problems easier than writing code from scratch*, "I like these order ones (Parsons problems), but I know, like at the end, there was always gonna be one where I actually had to write it, which is definitely more difficult for me."*

All of the teachers and undergraduate students we observed scrolled back up to look at the previous worked example when trying to solve the Parsons problems on the same web page. They seemed to be checking the syntax of some of the statements because the distractors required them to identify which of the two very similar blocks was correct. This provides evidence that the distractors made them focus on common problems like incorrect case, wrong method name, or missing colons at the end of a control structure.

Teacher T10 said, *"I think it's a nice step for students that have done Scratch as they're heading into textual language, that it's sort of blockish, that they have blocks they're dragging rather than having to type everything. Well, I'm starting to see that myself, that block language and written language are essentially the same thing only with graphics."* Solving Parsons problems may be particularly helpful for learners who start

out in a blocks based language like Scratch (Maloney et al., 2010) and then transition to a textual language like Python.

5.15 Log File Analysis

After the observational study, I added the 16 Parsons problems to the student and teacher versions of the ebook for the Advanced Placement (AP) Computer Science Principles (CSP) course. Problems one to eight were added to chapter five which was about using variable names and built-in functions and the rest were added to chapter ten which was about repetition (loops). The goal was to test the effectiveness of the adaptation process during normal ebook use. I analyzed the log file data from the student version of the AP CSP ebook from August 2017 to Jan 2018 to determine how many unique users attempted each problem, the percentage who got it correct, the number who used help, the percentage of those that used help that got the problem correct, and the percentage who didn't use help who got the problem correct. This data is shown in Table 19. Notice that more users attempt the first Parsons problem in each chapter (problems 1 and 9) than the last (problems 8 and 16).

Table 19. Log file analysis of the sixteen Parsons problems. The problems with an A after the problem id were adaptive. The others were not adaptive.

Problem Id	Number Users	Percent Correct	Number and % Used Help	Percent Correct if Used Help	Percent Correct if Didn't use Help	P value
1	108	37%			37%	
2-A	51	69%	7 (14%)	100%	64%	0.06
3-A	57	77%	14 (25%)	93%	72%	0.09
4	46	65%			65%	
5	48	63%			63%	
6-A	28	86%	3 (11%)	100%	84%	0.46
7-A	29	72%	8 (28%)	88%	67%	0.24
8	22	86%			86%	
9	101	58%			58%	
10-A	41	73%	8 (20%)	63%	76%	0.28
11-A	83	60%	25 (30%)	84%	50%	0.00*
12	68	63%			63%	
13	68	44%			44%	
14-A	66	47%	16 (24%)	81%	36%	0.02*
15-A	48	38%	13 (27%)	69%	26%	0.00*
16	41	29%			29%	

The percentage of people who got problems 1-8 correct if they used the help ranged from 88% to 100%. None of these problems required indentation. For problems 9-16, the percentage of people who got the problem correct after using help ranged from 63% to 84%. These problems all required indentation.

Looking at just the problems where adaptation was possible, (2, 3, 6, 7, 10, 11, 14, 15), we compared the group of users that used the adaptation to the group of users that did not use the adaptation. A Chi-squared test indicates a statistically significant

difference ($p < .001$) between the two groups on getting the problem correct. A Kruskal-Wallis test also indicated a statistically significant difference ($p < .001$). We then tested the difference between those two groups for each of the adaptive problems. We find a statistically significant difference using the Kruskal-Wallis test on problems 11, 14 and 15 as shown in Table 19. There was no statistically significant difference on any of the other adaptive problems, likely due to the low number of users who used the adaptation on those problems.

While these results show that a significantly higher percentage of the users who use the adaptation get the problems correct than those who do not use the adaptation for at least some of the problems, the adaptation can still be improved. Not all learners are successfully solving a problem when they use the adaptation. However, it may not be possible to reach 100% correct after receiving help. Some people may decide to reset the problem after getting help before they solve it, as seen in the observation of teacher T10 on problem 11.

5.16 Discussion

My hypotheses were:

- **H2A:** Most learners will understand the intra-problem adaptation process.
- **H3A:** A greater percentage of learners will correctly complete intra-problem adaptive Parsons problems than will complete non-adaptive Parsons problems.
- **H3B:** Most learners will prefer adaptive Parsons problems to non-adaptive Parsons problems.

- **H4A:** Most learners will perceive that solving Parsons problems with distractors helped them learn to fix code with similar errors and write similar code.

The learners understood most of the adaption process, so hypothesis **H2A** was partially supported. Students and teachers both noticed when distractor blocks moved from the solution area to the source area and were disabled. They also noticed when blocks were moved and combined. However, some teachers didn't notice when a distractor that wasn't used in the solution, i.e. was still in the source area on the left, was disabled. While the distractor block was highlighted and then slowly grayed out, their attention was on the solution area, so they didn't notice what was happening in the source area. After this study, the software was modified to only disable distractors that were actually used in the solution. Providing the indentation was also less successful. At least one teacher tried to indent blocks after the indentation had been provided, which shows that she didn't realize what had happened. Some teachers also didn't notice that the indentation gave clues as to which blocks should be inside of a loop and which should be outside. Novices likely do not have enough experience to realize that the indentation provides the clues to the structure of the problem. In the future, I will not provide the indentation during intra-problem adaptation and see if that affects the percentage of users who get the problem correct after using help.

This study provides initial evidence that the intra-problem adaptation process helped learners succeed, because all the teachers in the observational study were able to solve all of the intra-problem adaptive Parsons problems (those with a *Help Me* button).

In contrast, two teachers gave up on solving a Parsons problem that was not adaptive (without a *Help Me* button). However, some of the teachers asked me for help as well. If the problem was adaptive, I encouraged them to click the *Help Me* button to get help, but some of these teachers might have given up instead if I wasn't there to encourage them. The log file analysis also indicates that the users who used the adaptation were statistically significantly more likely to get the problem correct than those who did not use the adaptation, which supports **H3A**. However, this may be due to users giving up after less than three attempts to solve the problem, which is before help is available.

All but one of the teachers preferred the intra-problem adaptive Parsons problems to the non-adaptive Parsons problems, which supports **H3B**. However, several teachers were concerned that the students might overuse the help and didn't want it available on summative assessments (those for a grade). Teachers liked that the help wasn't available until the user had made at least three full attempts at a solution.

All the users perceived that solving Parsons problems with distractors helped them learn to fix code with errors and to write code, which supports **H4A**. The teachers said that the distractors helped them notice and identify common syntax problems like the incorrect case, missing parentheses, and missing colons. This is encouraging, because if they didn't perceive that the Parsons problems helped them, they would be less likely to try to solve them.

We used the observational study to also test our user interface changes for the indentation guidelines, the wrong indentation signifier, and the purple edges that signified a correct and distractor block pair. The user interface changes varied in effectiveness. The

indentation guidelines were effective at signifying to users that they could indent blocks, but some users indented blocks when they didn't need to. This is likely because some of the problems included a guideline, even if indention wasn't needed to solve that problem. After this study, we modified the software to only provide the guidelines when the problem required indentation. The incorrect indentation signifiers shown in Figure 36 were effective. All users were able to determine that they needed to further indent or unindent blocks when they saw the yellow highlights and arrows at the edges of the blocks. The signifiers for the pairing of the correct and distractor code blocks were not as effective. Several teachers said that they didn't notice the purple edge decorations or didn't realize that they meant that they should use only one of the two blocks in the solution. We plan to add labels such as *1a* and *1b* to the right side of the paired blocks and test those. Labels would make it easier for groups to discuss the blocks and would also indicate the pairing even after one of the blocks was moved. In another of our studies we used both the purple edge decorations and subgoal labels as comments to indicate the pairing. In that study very few students used both the correct and distractor block from a pair (B. J. Ericson et al., 2017).

5.17 Limitations

This chapter reports on an observational study of 11 teachers solving eight intra-problem adaptive and eight non-adaptive Parsons problems with both paired and un-paired distractors. While our results provide some evidence for the effectiveness of intra-problem adaptive Parsons problems, these numbers are too small to provide substantial proof. However, the log file analysis, provides evidence that the difference between the

percentage of people who got the problem correct after using the adaptation was significantly higher than the percentage of people who got the problem correct without using any adaptation. However, one drawback to using log file data is that we don't know why students failed to use the adaptation when it was available or why they gave up on solving a problem before getting it correct. This difference may mostly be due to most users giving up on solving the problem before help was available.

The learners in this study perceived that solving Parsons problems helped them also when they had to fix code with errors or write code. However, that perception may not be true. Further study needs to be done to verify that solving Parsons problems has a measurable effect on the ability to fix and write code.

5.18 Conclusion

This chapter reports on a within-subjects observational study of teachers solving intra-problem adaptive Parsons problems and non-adaptive Parsons problems. In intra-problem adaptation, the difficulty of the current problem can be dynamically reduced by disabling distractors, providing indentation, and combining blocks. This adaptation was only allowed after the user had attempted at least three full solutions to reduce the possibility of the user "gaming" the system. This paper provides evidence that the users understood some of the adaptation process such as moving a distractor from the solution back to the source and disabling it and combining blocks. However, several teachers did not understand the disabling of distractors that were not used in the solution and some teachers didn't seem to realize what providing indentation meant. The intra-problem adaptation process was successful in that all of the teachers were able to solve all of the

adaptive problems, while two teachers could not solve all of the non-adaptive problems. In addition, our log file analysis found that a significantly higher percentage of users who used the adaptation solved the problem than users who didn't use the adaptation. However, this could be due to users who gave up on solving the problem before help was available. Also, not all of the users who used the adaptation were able to solve the problem, so the process and user interface could be improved.

This chapter also provides evidence that our indentation incorrect signifiers were effective since all users were able to fix the indentation. However, our signifier to show that the correct code block and distractor code block were paired was not as effective, several teachers didn't realize that the blocks were paired, or that they should use one of the paired blocks in a solution.

All of the teachers and undergraduate students in our observations reported that solving Parsons problems with distractors helped them learn to fix code with errors and write code. While this is encouraging, further studies should be done to empirically test this perception. The next study was intended to test this perception.

CHAPTER 6. SOLVING ADAPTIVE PARSONS PROBLEMS VERSUS NON-ADAPTIVE PARSONS PROBLEMS AND WRITING CODE

The between-subjects study described in chapter four of solving Parsons problems versus fixing code versus writing the equivalent code provided the initial evidence that solving Parsons problems with distractors took significantly less time than fixing the same code with the same errors as the distractors or than writing the equivalent code. There were significant learning gains for all groups from pretest to immediate posttest and delayed posttest. These results provided evidence that solving Parsons problems might be a more efficient in terms of time to complete the practice problems, but just as effective with respect to learning gains, form of practice than fixing or writing code. However, the learning gains might not have been solely due to the practice condition, since the student might have learned from the review, answers to the practice questions, or from repeated exposure to the same or similar problems. That study would also have been improved by adding a control group that worked on off-task problems during the instructional period. This would have made it clearer that the learning gains were due to the instructional practice rather than a result of being tested on the same or similar questions.

The within-subjects observational study described in chapter five provided initial evidence that the intra-problem adaptation process could successfully help learners solve

Parsons problems. However, it had not tested the efficiency (completion time) or effectiveness (learning gains from pretest to posttests) for adaptive Parsons problems.

6.1 Research Questions

RQ5: What is the efficiency (with respect to completion time) and effectiveness (with respect to learning gains from pretest to posttests) of 1) solving adaptive (both intra-problem and inter-problem) Parsons problems versus 2) solving non-adaptive Parsons problems versus 3) writing the equivalent code versus 4) solving off-task adaptive Parsons problems.

6.2 Goals for the Study

This study intended to fix some of the issues with the prior study that compared the effectiveness and efficiency of solving Parsons problems versus fixing and writing code as well as test adaptive Parsons problems. It removed extra material, modified questions, and added a control group. In addition, it removed the fix code condition since the prior study had not shown a statistically significant difference in the time to solve fix code versus write code problems. I also removed the cognitive load survey since we hadn't found any significant difference between the self-reported cognitive load in the first study. It would probably be better to conduct a within-subjects study to test self-reported cognitive load.

I tested three hypotheses with this study.

- **H5A:** Learners who solve on-task (related to the pretest questions) adaptive and non-adaptive Parsons condition will finish the instructional problems significantly faster than the learners who write code.
- **H5B:** Learners who solve on-task adaptive Parsons problems with distractors will achieve similar learning gains from pretest to posttest than learners who solve on-task non-adaptive Parsons problems or learners who write the equivalent code.
- **H5C:** Learners who solve off-task (not related to the pretest questions) adaptive Parsons problems (the control group) will have lower learning gains than those who solve on-task problems.

6.3 Inter-Problem Adaptation

This study added inter-problem adaptation to the Parsons problem software. This means that the difficulty of the next Parsons problem can be modified to make it easier or harder depending on the learner's performance on the last Parsons problem. If the learner solved the last Parsons problem in only one attempt, then the next Parsons problem was made more difficult by un-pairing distractors (randomly mixing them in with the correct code) and by using all available distractors. If it took the learner four or five attempts to solve the last Parsons problem, then on the next problem the distractors would be shown paired with the correct code blocks. If it took the learner 6-7 attempts to solve the last problem, then 50% of the available distractors were removed and the remaining distractors were shown paired with the correct code blocks on the next problem. If it

took the learner 8 or more attempts to solve the last problem, then all distractors were removed from the next problem.

6.4 Study Design

This was a between-subjects design, with one pretest and two posttests. There were two sessions in the study. The first session was 2.5 hours and included consent, a demographic survey, pretest, instructional material, and an immediate posttest. The second posttest, which lasted one hour and was held one week later, was administered to measure retention of the instructional material.

The instructional material in the first session contained four worked-example and practice pairs. Students were randomly assigned to one of four practice conditions for the instructional material: 1) solving on-task adaptive Parsons problems with paired distractors, 2) solving on-task non-adaptive Parsons problems with distractors randomly mixed in with the correct code, 3) writing the equivalent code as the Parsons problems, or 4) a control group that solved off-task adaptive Parsons problems with paired distractors. The instructional practice condition was the independent variable. The dependent variables were the performance on the pretest and posttests and the time spent on each practice problem.

6.5 Study Procedure

This study consisted of two separate sessions one week apart. Both sessions were held in a closed classroom with all participants attending at the same time. Students were instructed to bring their laptops, and were provided with scratch paper and a pen. All of

the study materials were online and students were asked to only use those materials, even though they had access to the Internet. Proctors checked that the students were on task and not visiting other web sites.

In the first session, the procedure was 1) provide consent and randomly be placed into one of the four practice conditions, 2) complete the demographic survey, 3) complete the pretest, 4) complete four worked examples plus practice pairs, where the type of practice problem differed based on the condition, and 5) complete the immediate posttest. See Figure 52 for a diagram of the procedure.

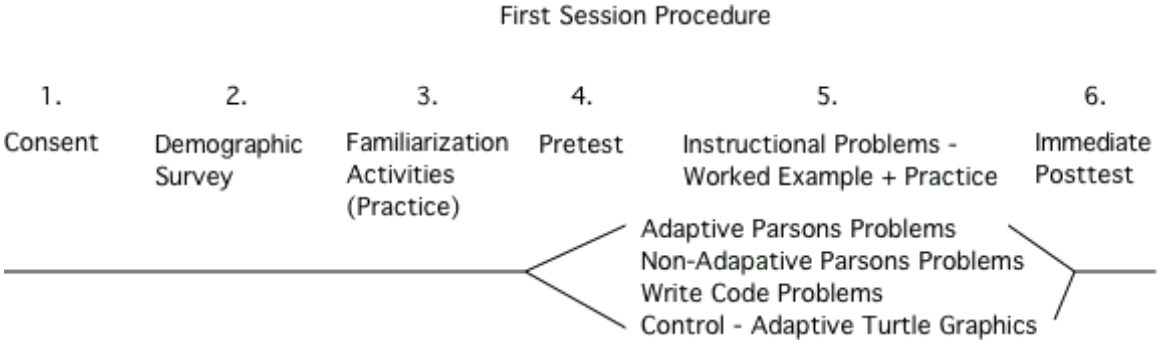


Figure 52. The first session procedure

At the second session, a week later, participants completed the delayed posttest, which was isomorphic to the first posttest. Only the variable names and some values were changed, but the structure of the problems was the same, meaning that they required near transfer to solve. Near transfer is being able to solve a new problem in a similar context to one that you have already solved. The delayed posttest tested for retention of the material one week later.

6.6 Study Materials

I reused most of the materials from the prior study described in chapter four. However, changes were made to the pretest and posttests to try to eliminate the ceiling effect on the multiple-choice questions and Parsons problem. Unlike the previous between-subjects study, I did not show students the correct answers after they submitted their answers to the instructional practice problems. This was to strengthen the claim that any learning gains were due to solving the four practice problems.

6.6.1 Demographic Survey

The demographic survey was the same as in the previous study described in chapter four. It asked for the participant's age, gender, race, first spoken language, comfort level with reading English, high school grade point average, college grade point average, current major, expected grade in the course, and prior programming experience. If they reported prior programming experience, they were also asked what courses and where they took them and how many years they had been programming. In addition, participants were asked to rate their ability to read, fix, and write Python code on a 5-point Likert scale.

6.6.2 Familiarization Material

The familiarization activities were also the same as in the chapter four. They included instruction on how to start and finish a timed exam, how to get to the next page, how to answer multiple-choice questions, how to check the solution for the fix code and

write code problems, and how to drag blocks and check the solution on a Parsons problem.

Also included were two easy practice multiple-choice questions, a practice fix code problem, a practice Parsons problem, and a practice write code problem. The fix code problem gave instructions to fix the code above the problem. The correct solution was displayed above the problem for the Parsons and write code problems.

6.6.3 Pretest

The structure and timing of the pretest remained the same as they were in the study from chapter four. There were four timed exams in the pretest. The participants had 15 minutes to complete the first timed exam of five multiple-choice questions and 10 minutes to complete each of the other three timed exams (fix code, Parsons problem, and write code).

In our previous study, there was a ceiling effect on the multiple-choice questions and the Parsons problem. While most of the multiple-choice questions in the pretest stayed the same as before, I replaced question two with a question from another study as shown in Figure 53 (Lister et al., 2004).

2-1-2: Consider the following code fragment. What will it print when it executes?

```
list1 = [1, 2, 4, 7]
list2 = [1, 2, 5, 7]
i1 = len(list1) - 1
i2 = len(list2) - 1
count = 0;
while i1 > 0 and i2 > 0:
if list1[i1] == list2[i2]:
    count=count + 1
    i1 = i1 - 1
    i2 = i2 - 1
elif list1[i1] < list2[i2]:
    i2 = i2 - 1
else:
    i1 = i1 - 1
print(count)
```

- A. 4
- B. 3
- C. 2
- D. 1
- E. 0

Figure 53. The new pretest multiple-choice question #2. The answer is C since the while loop stops before i1 and i2 reach 0.

I also modified the answers to question number four shown in Figure 54 based on answers the students had calculated on their scratch paper (Cunningham et al., 2017) during the previous study. Since these are answers that students actually calculated they should be better distractors than the prior answers.

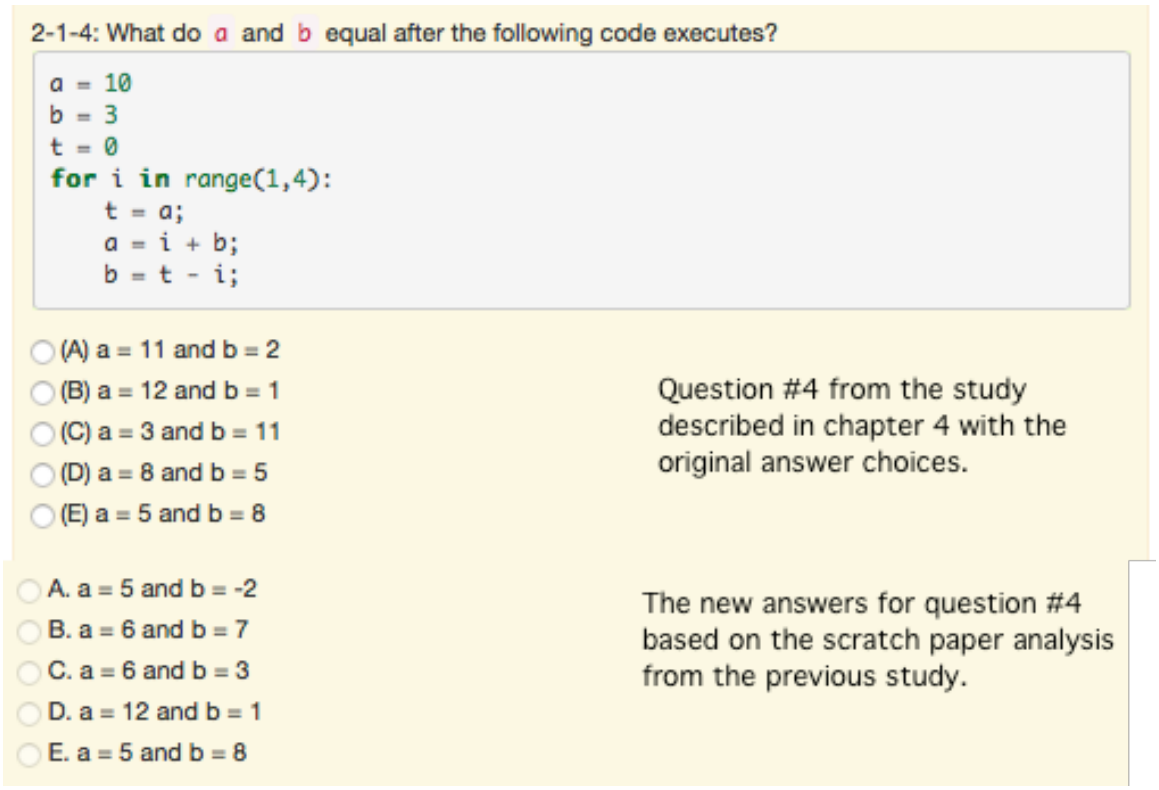


Figure 54. The pretest multiple-choice question #4 with the old and new answers. The correct answer is E.

The second timed exam contained one fix code problem. This was a new problem that was not used in the prior study. The code was intended to calculate and return the average of a list of numbers, but double the highest value. However, it had errors that the learner had to fix as shown in Figure 55.

```
def getAvgDoubleHighest(nums):

    #if no value in list return 0
    if len(nums) = 0:
        return 0

    # initialize variables
    sum = 0
    max = 0

    # loop through indicies
    for index in range(nums):

        # get value and add to sum
        value = nums[index]
        sum = sum + value

    # check for new max
    if value < max:
        max = value

    # return average with doubled mc
    sum = sum + max
    return sum / len(nums)
```

```
def getAvgDoubleHighest(nums):

    #if no value in list return 0
    if len(nums) == 0:
        return 0

    # initialize variables
    sum = 0
    max = nums[0]

    # loop through indicies
    for index in range(len(nums)):

        # get value and add to sum
        value = nums[index]
        sum = sum + value

        # check for new max
        if value > max:
            max = value

    # return average with doubled max
    sum = sum + max
    return sum / (len(nums) + 1)
```

Figure 55. Pretest fix problem with the errors on the left boxed and a correct solution on the right.

The problem also contained four unit tests to test the learner's solution as shown in Figure 56. Each unit test showed the input, the expected output, and the actual output, and whether the test passed or failed.

Result	Actual Value	Expected Value	Notes
Pass	2	2	Test of getAvgDoubleHighest([1,1,3])
Pass	0	0	Test of getAvgDoubleHighest([])
Pass	3.0	3.0	Test of getAvgDoubleHighest([3.0,5.0,2.0,0]
Pass	-3.5	-3.5	Test of getAvgDoubleHighest([-3,-3,-5]

You passed: 100.0% of the tests

Figure 56. Unit test results on the fix code problem when it is correct.

The Parsons problem was the same as the write code problem from the previous study. It returned true if the maximum value minus the minimum value in a list was less

than or equal to 10. It had five un-paired distractors randomly mixed in with the correct code blocks as shown in Figure 57.

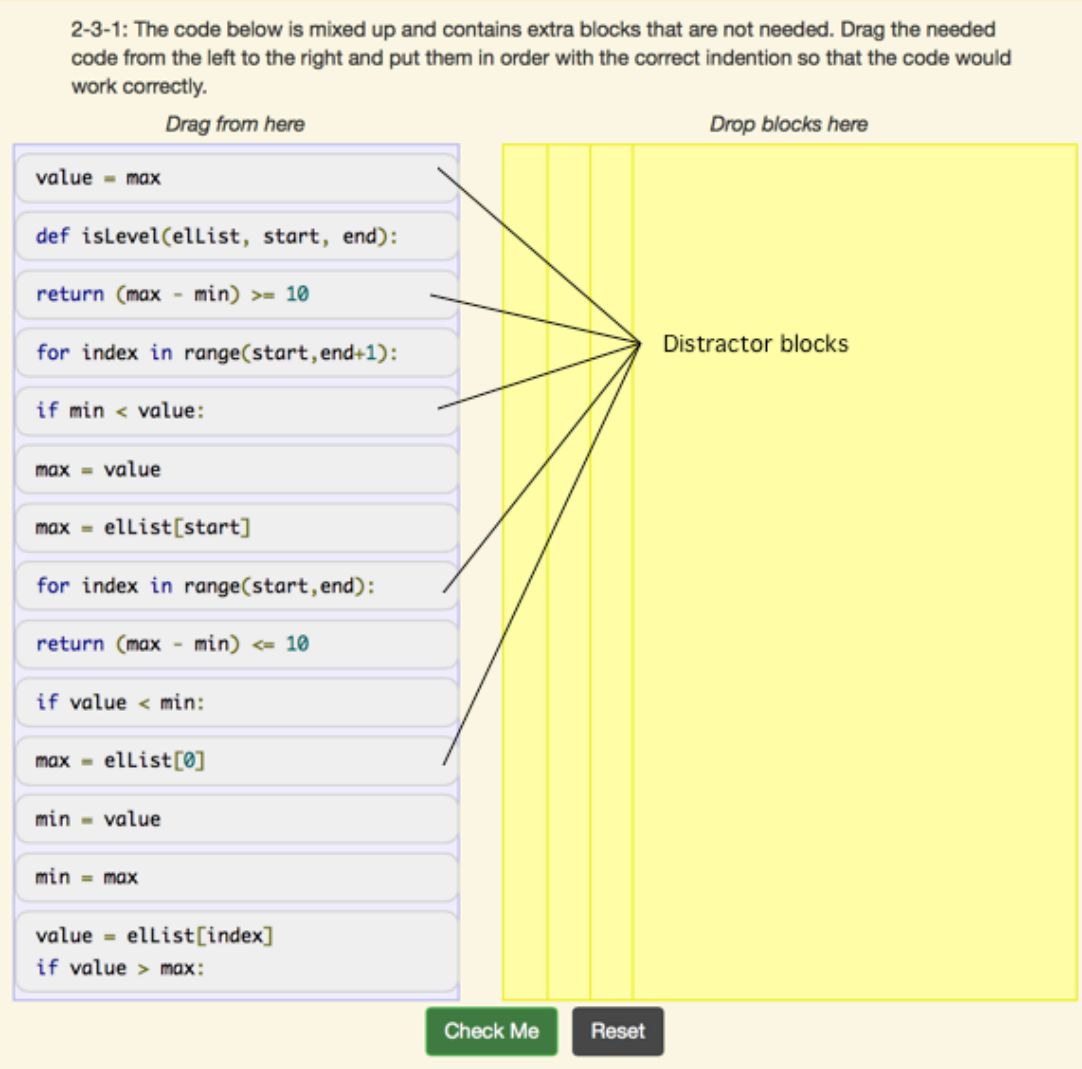


Figure 57. Pretest order code (Parsons) problem with five distractors.

The distractors are shown on the left in the source area and the solution is shown on the right in the solution area in Figure 58. When you create a Parsons problem you have to break up the correct statements into code blocks in a way that ensures that there is only one correct solution. While it doesn't really matter if you check if the current value

is greater than the maximum value or less than the minimum value first, I kept the test if the value is greater than the maximum value with the statement above it in one code block as shown in Figure 58 to force the order.

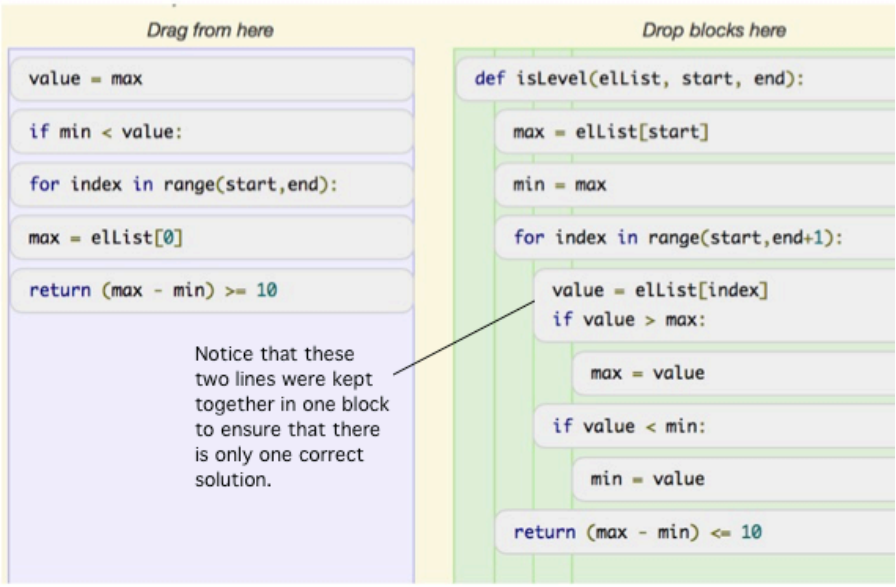


Figure 58. The distractors on the left side and the correct solution on the right side.

The write code problem was the same as the fix code problem from the previous study. It was a modified version of Soloway's rainfall problem which has been extensively studied (Simon, 2013; Soloway, 1986). The original problem totals the non-negative values in an input loop until a sentinel value is reached and then outputs the average. The solution must avoid a division by zero. The problem was modified to loop through a list of numbers rather than read input until a sentinel value was reached. Simon found that students still perform poorly on this problem and that students are not used to reading input in a loop until a sentinel value is reached (Simon, 2013). The instructions explained the algorithm in English, provided example input and output, and provided hidden unit tests. A correct solution and the passed unit tests are shown in Figure 59.

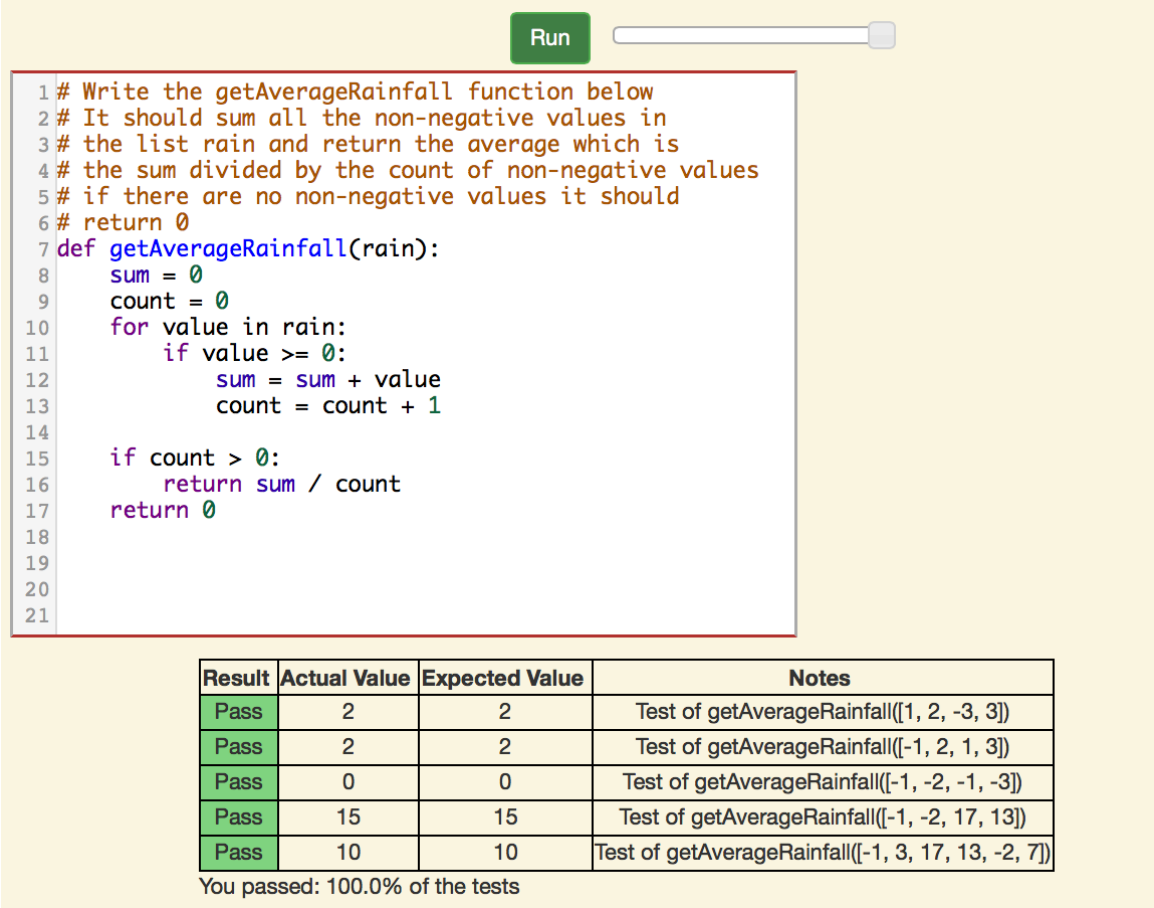


Figure 59. A correct solution to the pretest write code problem (the rainfall problem).

6.6.4 Instructional Material

The instruction material contained four worked examples with interleaved practice problems. The worked examples contained an algorithm in English and example input and output as shown in Figure 60, as well as runnable Python code with hidden unit tests as shown in Figure 61.

Example 3: Find the Average of a List of Values

Another common thing to do with a list of values is find the average. To prevent a divide by zero you can first check if the length of the list is 0 and if so return 0. Otherwise, create a variable to hold the sum and initialize it to 0. Then loop through all the indices in the list and add the value at the current index to the sum. Return the sum divided by the number of items in the list (the length of the list).

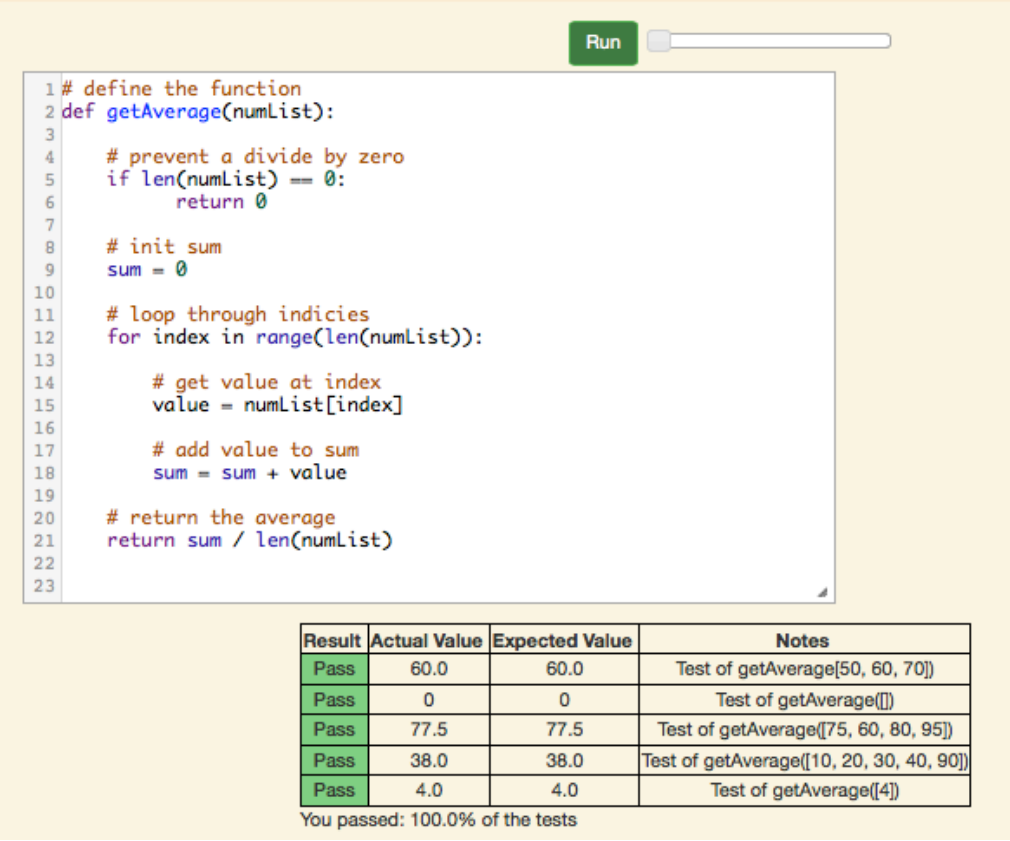
Examples

For example `getAverage([50, 60, 70])` should return 60 and `getAverage([])` should return 0.

Run Code

Click the Run button to run the tests that check that this code is working correctly. All tests should print Pass since this is correct code. Scroll down to try to solve the practice problem below.

Figure 60. Worked example #3 with the algorithm in English and example input and output.



Result	Actual Value	Expected Value	Notes
Pass	60.0	60.0	Test of getAverage[50, 60, 70])
Pass	0	0	Test of getAverage([])
Pass	77.5	77.5	Test of getAverage([75, 60, 80, 95])
Pass	38.0	38.0	Test of getAverage([10, 20, 30, 40, 90])
Pass	4.0	4.0	Test of getAverage([4])

You passed: 100.0% of the tests

Figure 61. Worked example #3 code and the results from running the unit tests.

The practice problems varied by condition with one group solving adaptive two-dimensional Parsons problems with paired distractors, one solving non-adaptive two-dimensional Parsons problems with distractors randomly mixed in with the correct code, the third writing the equivalent code, and the control group solving off-task adaptive turtle graphics two-dimensional Parsons problems with paired distractors. Each of the practice problems also contained an algorithm in English, example input and output, and a way to test the solution. Each practice problem was in a timed exam and each had a time limit of 10 minutes.

The first worked example in the first three conditions returned a count of the number of times a target value appeared in a list using a loop that looped through all the indices. The associated practice question was to return the count of a target value in a given range of indices (inclusive). The second worked example returned the maximum value from a list and the associated practice problem was to return the minimum value. The third worked example returned the average of the values in a list and protected against a divide by zero error as shown in Figure 61. The associated practice problem returned the average, but didn't include the lowest value in the list in the average and also guarded against a divide by zero error as shown in Figure 62. The fourth worked example returned the minimum value in a given range of indices (inclusive). The associated practice problem returned the maximum value in a given range of indices (inclusive).

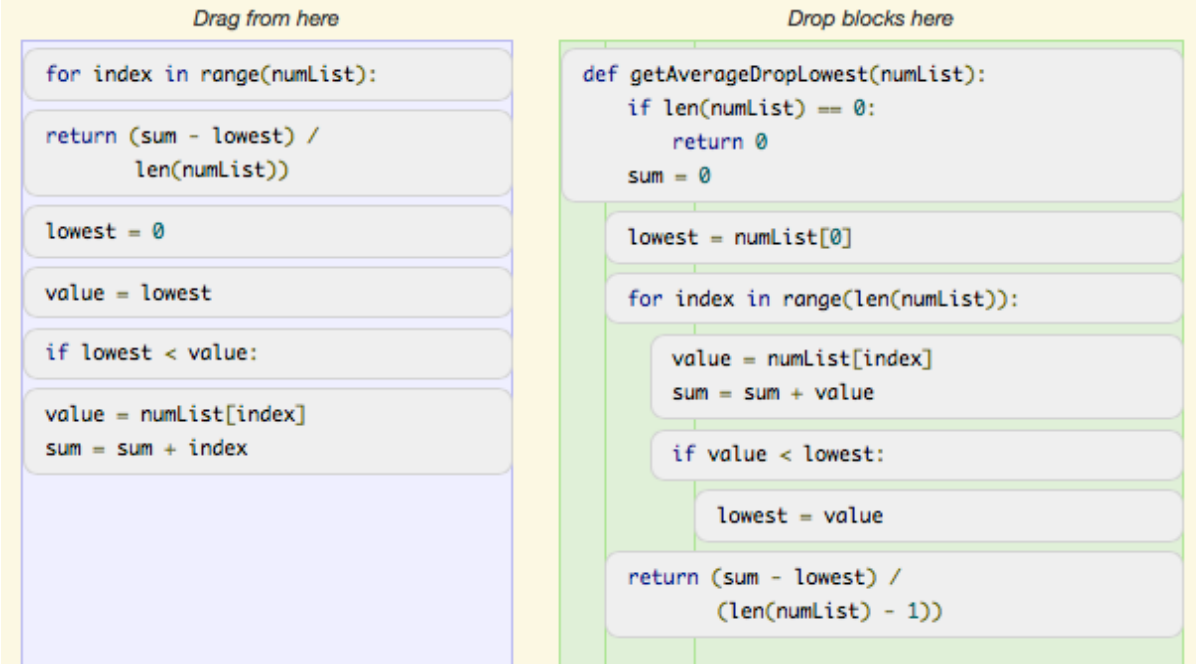


Figure 62. Distractors on the left and the answer on the right for practice problem #3 in the non-adaptive Parsons condition.

6.6.5 Posttests

The immediate posttest in the first session had the exact same questions as the pretest. The delayed posttest, which was administered one week later, was isomorphic to the immediate posttest, meaning that the problems to be solved had the same structure, but different surface level features, like variable names as shown in Figure 63.

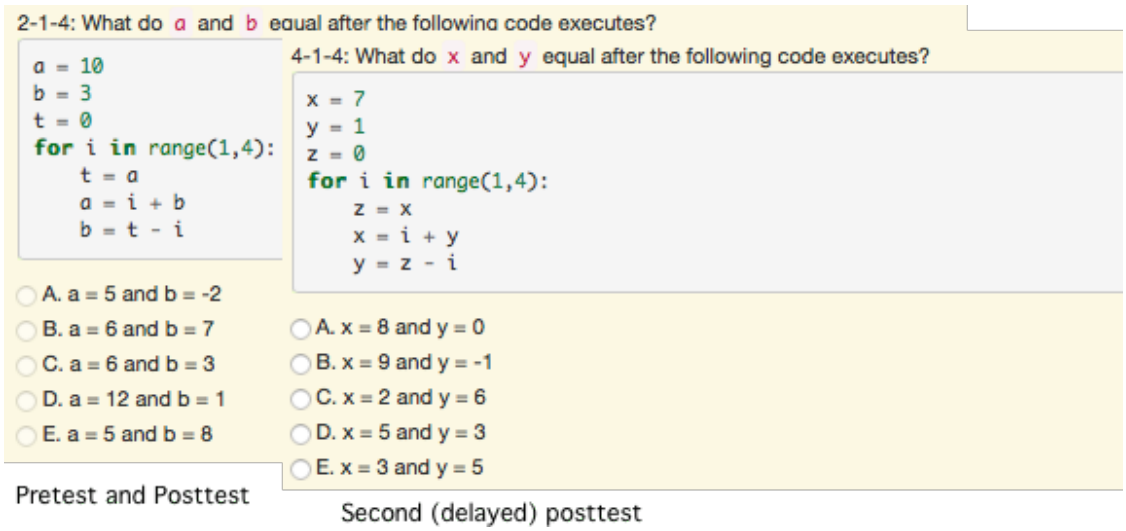


Figure 63. The #4 multiple choice question from pretest (left) and second posttest (right). Notice that the variable names and values have changed.

6.7 Participants

Undergraduate students were recruited from two sections of a first computer science course for computing majors at the Georgia Institute of Technology, a research-intensive university in the United States. The sections had the same instructor and followed the same curriculum with the same homework and assessments. This course covers introductory programming concepts in Python including variables, selection, iteration, and lists. At the time of the study the course had covered all of these topics and was covering files and dictionaries. I visited the course during lecture to recruit participants and also sent an announcement to all of the students enrolled in the course. Participants could earn 2.5 points of extra credit for completing the first session and another 2.5 points of extra credit for completing the second session one week later. Students who did not participate in the pilot study or large-scale study could alternatively earn up to 5 points of extra credit by writing a paper on a computing innovation, which I

graded and that grade was submitted to the course instructors. I was not involved in the teaching of the course.

6.8 Analysis

A total of 163 students participated in the first session. However, 37 of these students did not answer at least one question during the session or spent less than 30 seconds answering a question without getting the question correct. I am reporting on the data from 126 students (32 in the adaptive Parsons condition, 34 in the non-adaptive Parsons condition, 27 in the write condition, and 33 in the control group that solved off-task adaptive Parsons problems) from the first session. Students were not required to come back for the second session one week later, but earned an additional 2.5 points of extra credit for completing that session. A total of 126 students returned for the second session. Of these, 100 students completed all the questions in both the first session and second session and spent at least 30 seconds on each question or got the question correct in under 30 seconds (27 in the adaptive Parsons condition, 30 in the non-adaptive Parsons condition, 19 in the write condition, and 24 in the control group that solved off-task adaptive Parsons problems).

6.8.1 Testing for Efficiency

For each of the four instructional practice problems, the elapsed time in seconds was calculated from the start and end time on the timed exams, to compare the efficiency of the conditions. The mean time and standard deviation by condition for each of the four practice problems is shown in Table 20. Note that the adaptive Parsons and non-

adaptive Parsons had similar mean completion times. In my observational study of teachers solving both adaptive and non-adaptive Parsons problems, sometimes the adaptive problems took longer to solve than the non-adaptive because the teachers kept checking their solution or thinking about what to do next. With the inter-problem adaptation, if the learner struggled on the previous problem then the next problem is made easier and if the learner solved the previous problem in one attempt the next problem is made harder, which probably kept the total completion times similar. Also note that the write code and control conditions (off-task adaptive Parsons problems) had longer completion times than the on-task adaptive and non-adaptive Parsons problems.

Table 20. The mean time in seconds and standard deviation for each of the four practice problems by condition.

Group	Practice 1 Mean in Seconds (std dev)	Practice 2 Mean in Seconds (std dev)	Practice 3 Mean in Seconds (std dev)	Practice 4 Mean in Seconds (std dev)
Adaptive Parsons (n=32)	115.65 (50.1)	97.88 (34.3)	191.88 (130.5)	74.63 (23.5)
Parsons (n=34)	114.29 (56.3)	92.85 (31.0)	190.79 (91.2)	72.94 (26.8)
Write (n=27)	177.44 (152.0)	118.07 (113.3)	270.48 (152.0)	102 (63.6)
Control (n=33)	252.24 (100.9)	176.70 (79.6)	178.12 (107.13)	325.06 (160.3)

To test if the time differences were significant, outliers were removed (values more than three standard deviations from the mean) to normalize the data so that z-scores could be calculated. Z-scores allow for different size groups to be compared. A test for skew (a test to indicate whether or not the data falls in a normal distribution) revealed that

the values were all in the acceptable range (under 2). Removing outliers left 31 students in the adaptive Parsons group, 33 in the non-adaptive Parsons group, and 22 in the write group. Z-scores were created from the total time in seconds to solve the four practice problems minus the mean and divided by the standard deviation. A Least Squares Difference test (LSD) was used to compare the three groups (Adaptive Parsons, Parsons, and Write). The fourth condition was the control group, which was solving off-task problems, so it was not included in the test for efficiency. There was a statistically significant difference between the groups with an effect size of .073 - a medium effect size. There was no significant difference in completion time between the adaptive Parsons group and non-adaptive Parsons group. The time was significantly different between the adaptive Parsons group and the write group (mean difference of -.33 and p=.025) as well as the non-adaptive Parsons group and the write group (mean difference of -.32 and p=.025).

6.8.1 Testing for Effectiveness

I created grading rubrics for the write and fix code problems on the pretest and posttests. Two people graded each problem independently and then met to resolve any differences in scores. The hand graded scores correlated with the unit test results. A unit test checks that the expected output from a function matches the actual output. The fix code problem had four unit tests and the write code problem had five unit tests. A factor analysis showed that the hand graded scores and unit test scores appeared to be measuring the same construct.

I automated the grading for the Parsons problems. Grading started from the beginning of the solution and each correct line in the proper order received a half point and if the line or its paired distractor was indented correctly it received half a point. Grading continued until a line was found that was neither the correct line nor its paired distractor (i.e. a line out of order). Grading then continued from the end of the solution in the same fashion toward the first line that had been found to be incorrect. This grading approach was based on my observation that learners had the most difficulty in the middle of the solution. I also wanted the grading to be similar to the grading of the fix code problems, and the fix code problems had the advantage that the code was already in the correct order.

The mean and standard deviation for each pretest and immediate posttest timed exam are shown by condition in Table 21. The pretest and posttest both contained four timed exams. One timed exam had five multiple-choice (MC) questions, one had a fix code problem, one had a Parsons problem, and one had a write code problem.

Table 21. Mean and (Standard Deviation) for each timed exam on the pretest and immediate (1st) posttest by condition.

	Adaptive Parsons (n=32)	Non-Adaptive Parsons (n=34)	Write (n=27)	Control (n=33)
Pre MC (max 5)	2.7 (1.5)	3 (1.1)	3.8 (1.4)	3.6 (1.4)
Post MC	3.8 (1.1)	3.4 (.17)	4.3 (1.3)	4.2 (1.2)
Pre Fix (max 11)	8.1 (1.6)	8.9 (2.0)	9.0 (1.6)	8.8 (1.8)
Post Fix	9.2 (1.8)	9.6 (2.0)	9.8 (1.8)	8.8 (2.1)
Pre Parsons (max 10)	7.3 (3.3)	8.6 (3.0)	7.7 (3.4)	7.4 (3.6)
Post Parsons	8.5 (3.0)	9.5 (1.8)	8.0 (3.3)	7.9 (3.5)
Pre Write (max 10)	8.6 (2.3)	9.3 (1.3)	9.0 (1.7)	9.2 (1.2)
Post Write	9.3 (1.3)	9.4 (1.0)	9.0 (1.9)	9.2 (1.4)

Remember that not all of the students took the delayed (2nd) posttest one week later. The mean score and standard deviation are shown for the pretest, immediate posttest, and delayed (2nd) posttest for the just the students who attended both sessions is shown in Table 22.

Table 22. Mean and (Standard Deviation) for each timed exam on the pretest, immediate posttest, and delayed posttest (2nd posttest) by condition.

	Adaptive Parsons (n=27)	Non-Adaptive Parsons (n=30)	Write (n=19)	Control (n=24)
Pre MC (max 5)	2.7 (1.5)	2.9 (1.1)	3.8 (1.3)	3.8 (1.5)
Post MC	3.9 (1.0)	3.7 (1.6)	4.2 (1.5)	4.3 (1.3)
2nd Post MC	3.8 (1.1)	3.7 (1.2)	4.3 (1.1)	3.8 (1.2)
Pre Fix (max 11)	8.1 (1.6)	8.9 (2.0)	8.9 (1.6)	8.6 (1.9)
Post Fix	9.3 (1.7)	9.6 (2.0)	9.4 (2.0)	8.8 (2.2)
2nd Post Fix	9.1 (1.8)	9.7 (1.9)	9.5 (1.8)	9.3 (1.6)
Pre Parsons (max 10)	7.7 (3.1)	8.1 (3.3)	6.8 (3.7)	7.2 (3.8)
Post Parsons	8.2 (3.1)	9.4 (2.0)	7.1 (3.7)	7.4 (3.7)
2nd Post Parsons	8.7 (2.4)	9.7 (1.4)	9.2 (2.0)	8.3 (2.9)
Pre Write (max 10)	8.9 (1.9)	9.4 (1.4)	8.7 (2.0)	9.1 (1.3)
Post Write	9.4 (1.2)	9.5 (1.1)	8.7 (2.2)	9.0 (1.5)
2nd Post Write	9.3 (1.1)	9.7 (1.0)	9.0 (2.1)	9.3 (1.3)

There was a statistically significant change from pretest to the immediate posttest using a multivariate analysis of variance (MANOVA) with Pillai's trace (F=2.36 and p=.031). A MANOVA was used since there were four conditions in this study (three with one control group). The Bonferroni post-hoc test doesn't indicate a statistically significant difference by condition from the pretest to the immediate posttest, which means that no condition seemed to have done better or worse than the others.

I also checked the difference in the scores from pretest to immediate posttest for all the on-task conditions compared to the control group. Twenty-seven results were chosen at random from each condition in order to compare equal size samples. A Mann-Whitney U test was used, which does not assume that the data follows a normal distribution. The difference between the pretest score and the posttest score was

significant (p = .007882) for the adaptive condition versus the control group (who solved turtle graphics problems). There was not a significant difference for any of the other on-task conditions compared to the control group. See Figure 64 for the box and whisker plot by condition.

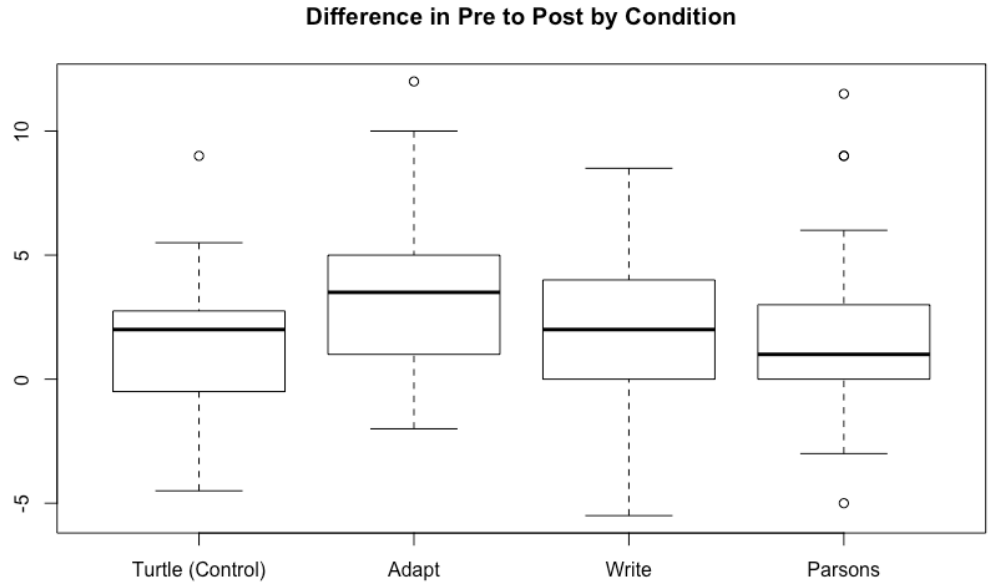


Figure 64. The difference between the immediate posttest composite score and the pretest composite score by condition

Kurtosis is high on the pretest write and posttest write problems which indicates that the scores fall in a narrow range. This means there was likely a ceiling effect on the write code problem. The mean scores on the pretest write problem ranged from 8.7 to 9.4 out of a maximum of 10 as shown in Table 22.

6.8.2 Use of intra-problem adaptation

None of the students in the adaptive Parsons condition used the intra-problem adaptation on problems 1 or 4 as shown in Table 23. A few students used the intra-problem adaptation on problems two and three. Only one person in the adaptive condition failed to correctly solve a problem, but that student didn't use the intra-problem adaptation. That person made six attempts total, so an alert would have been shown saying that help was available after the third incorrect attempts, but that student did not click on the *Help Me* button to initiate the intra-problem adaptation. That student had not used the help on either of the first two problems. I don't know if the student didn't read or understand the alert, or choose not to use the help.

Students also used the intra-problem adaptation in the off-task control group as shown in Table 23. It is encouraging to see that the students in the control (Turtle) group who used the intra-problem adaptation all solved problems one and two. Nearly all the students in the control (Turtle) group who used the intra-problem adaptation solved problem three. One student ran out of time. Two students didn't solve problem four. One of them also ran out of time.

Table 23. Number of students who got the problem correct out of the number who attempted the problem and the percent correct. Number of students who used the help (intra-problem adaptation) and got the problem correct out of those who used the help and the percent correct.

	# Correct / # Attempted and % Correct	# Correct & Used Help / # Used Help and % Correct when Used Help
Practice #1		
Adaptive	32/32 100%	No student used the help
Parsons	33/34 97%	
Write	23/27 85%	
Turtle	32/33 96%	11/11 100%
Practice #2		
Adaptive	32/32 100%	2/2 100%
Parsons	34/34 100%	
Write	25/27 93%	
Turtle	33/33 100%	8/8 100%
Practice #3		
Adaptive	31/32 97%	5/5 100%
Parsons	32/34 94%	
Write	21/27 78%	
Turtle	32/33 97%	8/9 89% (1 ran out of time)
Practice #4		
Adaptive	32/32 100%	No student used the help
Parsons	34/34 100%	
Write	26/27 96%	
Turtle	28/33 85%	15/17 88% (1 ran out of time)

6.8.3 Use of inter-problem adaptation

Inter-problem adaption can change the difficulty of the *next* problem based on the learner's performance on the *previous* problem. If a student solves the previous problem in just one attempt, the next problem will use all distractors and un-pair them (mix them in randomly with the correct blocks). If it takes two to three attempts to solve the current problem, there is no change to the next problem. If it takes four to five attempts to solve

the current problem, the next problem will pair distractors with the correct code blocks. If it takes six to seven attempts to solve the current problem, the next problem will use half of the available distractors and will pair them with the correct code blocks. If it takes eight or more attempts to solve the current problem, the next problem will not include any distractors. See Table 24 for the number and percentage of students who solved each instructional adaptive Parsons problem in the specified number of attempts. For example, the students who solved problem one in just one attempt (15 students) then solved problem two with all distractors randomly mixed in with the correct code (un-paired), whereas the students who solved problem one in four to five attempts solved problem two with paired distractor and correct code blocks. Over 50% of the students in the adaptive Parsons condition solved problem three with un-paired distractors since they solved problem two in just one attempt. Thirty-two percent of the students solved an easier version of problem four because it took them four or more attempts to solve problem three.

Table 24. Number and percentage of students who solved each instructional adaptive Parsons problem in that number of attempts

Problem	1 attempt	2-3 attempts	4-5 attempts	6-7 attempts	8+ attempts
1	15 (47%)	15 (47%)	2 (6%)	0	0
2	17 (53%)	12 (38%)	1 (3%)	0	2 (6%)
3	12 (38%)	10 (31%)	4 (13%)	1 (3%)	5 (16%)
4	19 (59%)	11 (34%)	2 (6%)	0	0
On next problem:	**Un-pair & use all distractors**	**No change**	**Pair distractors**	**Remove 50% of distractors**	**Remove all distractors**

6.8.4 Analysis of the demographic information

Of the 126 students who completed all questions in the first session, 73 (58%) self-identified as male and 51 (40%) as female and two (2%) students didn't answer the question. Three (2%) students identified as Hispanic, eight (6%) as Black or African American, 53 (42%) as Asian, and 62 (49%) as White and three (2%) as other (Middle Eastern). Students could pick more than one category, so the total is more than 100%. Ninety-nine (79%) of the students reported that English was their first language, 17 (13%) reported Korean, six (5%) Mandarin, and one (1%) Spanish. Most (60%) of the students reported their age as 18 as shown in Table 25.

Table 25. The number and percentage of students by age

Age	**17**	**18**	**19**	**20**	**21**	**22**	**23-29**	**30-39**
# - %	8 - 6%	76 - 60%	20 - 16%	11 – 9%	6 – 5%	1 – 1%	2 – 2%	1 – 1%

This is a computer science course that was created for computer science majors, but the majority of the students in the study were non-majors as seen in Table 26.

Table 26. The student's majors in the first session of the study

Major	**Number (Percent of Total)**
Computer Science	39 (31%)
Engineering	27 (21%)
Mathematics	14 (11%)
Business	8 (6%)
Physics	7 (6%)
Economics	6 (5%)
Computational Media	5 (4%)
Biology	5 (4%)
Other	15 (12%)

We did an analysis of the demographic information versus the pretest, immediate posttest and delayed posttest results. All had a strong positive correlation with the student's final grade in the course. The pretest correlation was $r(124) = .597$, $p < .001$, the immediate posttest was $r(124) = .520$, $p << .001$, and the delayed posttest was $r(99) = .666$, $p < .001$. We had a moderate negative correlation between the pretest score and the student's age $r(121) = -.413$, $p < .001$, which means that older students did worse than younger. This course is intended to be a first course for majors, so older students may be retaking the course after failing it in the past, or be weaker students who delayed taking the course. We found a moderate negative correlation for gender with males performing better than females on the delayed posttest $\rho(99) = -.362$, $p < .001$. There was also a moderate positive correlation for prior programming experience, with those who had prior experience doing better than those who did not on the pretest $\rho(125) = .231$, $p = .010$ and immediate posttest $\rho(125) = .244$, $p = .006$.

We found no interaction between condition and any of the demographic characteristics that affected performance. This means that the groups were comparable.

6.9 Discussion

My hypotheses were:

- **H5A:** Learners who solve on-task (related to the pretest questions) adaptive and non-adaptive Parsons condition will finish the instructional problems significantly faster than the learners who write code.
- **H5B:** Learners who solve on-task adaptive Parsons problems with distractors will achieve similar learning gains from pretest to posttest than

learners who solve on-task non-adaptive Parsons problems or learners who write the equivalent code.

- **H5C:** Learners who solve off-task (not related to the pretest questions) adaptive Parsons problems (the control group) will have lower learning gains than those who solve on-task problems.

Both on-task Parsons groups (the adaptive and non-adaptive) solved the four practice problems in significantly less time than the write code group. This supports hypothesis **H5A**.

There was a significant improvement in composite scores from pretest to immediate posttest. However, there was no significant difference between the three on-task conditions. This means that learners solving both adaptive and non-adaptive Parsons problems had equivalent learning gains as those in the write code condition. This supports hypothesis **H5B**.

In addition, the students in the adaptive Parsons condition who used the intra-problem adaptation (clicked the *Help Me* button) had a higher success rate than the students in the write code condition for each of the four practice problems. This indicates that the adaptation process did successfully help students correctly complete the problem. This is important because successful practice should lead to more learning than unsuccessful practice.

The learners in the control group who solved off-task turtle graphics adaptive Parsons problems had a significantly lower learning gain from pretest to immediate posttest than the on-task adaptive Parsons group, which supports hypothesis **H5C**.

However, there was no significant difference between the control group and the non-adaptive Parsons group or between the control group and the write code group. This means that hypothesis **H5C** was not fully supported. This indicates that at least part of the learning gain from pretest to immediate posttest was due to repeated practice on the same problems.

6.10 Limitations

The intra-problem adaptation (which was initiated by clicking the *Help Me* button) was not extensively used in the adaptive Parsons condition. No students in that condition used the intra-problem adaption on problem one or four, only two (6%) students used it on problem two, and five (16%) used it on problem three. The inter-problem adaptation had more of an effect on the difficulty of the problems. The inter-problem adaptation made problem three harder for over 50% of the students and made problem three easier for over 30% of the students. However, it isn't clear how much each type of adaptation contributed to the resulting learning gains from pretest to posttest. More studies need to be done to gauge the effect of each type of adaptation.

The significant difference between the control group and the adaptive Parsons group for learning gains from pretest to immediate posttest, does support the idea that solving adaptive Parsons problems can lead to learning gains. However, the fact that there was no statistically significant difference between the control group and the non-adaptive Parsons group or the write code group in terms of learning gains from pretest to posttest means that at least some of the learning gain may be due to repeated practice with the same or similar questions. More research needs to be done to verify the

effectiveness in terms of learning gains from pretest to posttests of solving adaptive Parsons problems compared to non-adaptive Parsons problems and/or writing the equivalent code. The control group used the intra-problem adaptation more than the adaptive Parsons group, which suggests that the turtle graphics problems solved by the control group may be of the appropriate level of difficulty for a future study.

6.11 Conclusion

This chapter reports on a between-subjects study comparing the efficiency and effectiveness of adaptive Parsons problems, non-adaptive Parsons problems, and writing the equivalent code. The adaptive Parsons problems in this study included both intra-problem and inter-problem adaptation. In intra-problem adaptation if the leaner is struggling to solve the current problem then that problem can be dynamically made easier by disabling distractors, providing indentation, and/or combining code blocks. In inter-problem adaptation if the learner struggled to solve the last problem the next problem is made easier by paring distractors or removing distractors. If the learner solved the current problem easily then the next problem can be made harder by using randomly mixed in distractors instead of paired distractors or by adding distractors.

There was a significant difference in the completion time for both adaptive and non-adaptive Parsons problem groups compared to writing code group. This provides evidence that solving adaptive and non-adaptive Parsons problems is a more efficient form of practice than writing the equivalent code. There was a significant difference from pretest to immediate posttest for the adaptive Parsons problem group compared to the control group, which provides some evidence of learning from solving adaptive

Parsons problems. However, there was no significant difference in learning gains between the control group and either the non-adaptive Parsons problems group or the write code group. This means that at least some of the learning gains are likely due to repeated exposure to the same problem from pretest to posttest. Further studies need to be done to test the learning gains from solving adaptive Parsons problems versus non-adaptive Parsons problems versus writing the equivalent code.

CHAPTER 7. CONTRIBUTIONS AND FUTURE WORK

This thesis contributes to the research on Parsons problems and adaptive learning. It investigated the efficiency with respect to the time that it takes to complete practice problems and the effectiveness with respect to learning gains from pretest to immediate posttest and delayed posttest. It compared solving Parsons problems with distractors, to fixing code with the same errors as the distractors, to writing the equivalent code. It also compared solving intra-problem and inter-problem adaptive Parsons problems, to solving non-adaptive Parsons problems, as well as to writing the equivalent code.

In intra-problem adaptation, the current problem can dynamically be made easier by disabling distractors, providing indentation, or combining blocks. In inter-problem adaptation, the difficulty of the next problem is based on the learner's performance on the previous problem. If the learner solved the previous problem in one attempt, the next problem is made harder by un-pairing the distractors. If the learner struggled to solve the previous problem, the next problem can be made easier by pairing or removing distractors. This research used a mixed-methods approach with observational studies, log file analyses, and experiments. This chapter summarizes the findings from each of my research studies. It then describes my short-term and long-term future work.

7.1 Summary of Initial Investigations into Parsons Problems

The preliminary observational study with four teachers solving 11 Parsons problems showed that the teachers could solve the Parsons problems in one or two attempts (B. J. Ericson et al., 2015). According to Parsons and Haden, Parsons problems

are at the right level of difficulty if they are solvable in two to three attempts (Parsons & Haden, 2006). While they don't explain their reasoning, it is likely to reduce learner frustration. The teachers had the most trouble with Parsons problems that required indention, which is consistent with prior research that showed that Parsons problems that require indentation (two dimensional Parsons problems) are harder to solve than those that do not (Denny et al., 2008; Ihantola & Karavirta, 2011). Teachers thought that the Parsons problems were interesting and helped them learn the order of the statements in a turtle graphics program, but some teachers wanted the problems to be more challenging. However, these teachers had more experience than most novice programmers.

A log file analysis of students solving the same Parsons problems provided evidence that some of the Parsons problems were too easy, some were about the right level of difficulty, and some were too hard (B. J. Ericson et al., 2015). While most students correctly solved the Parsons problems eventually, some struggled and took much longer than expected to solve them (over 100 attempts). Some students gave up and never solved some of the problems. The log file analysis also showed that more students attempted Parsons problems then attempted nearby multiple-choice questions, which encouraged me to investigate Parsons problems as a type of low-cognitive load practice problem.

7.2 Summary of Solving Parsons Problems Versus Fixing and Writing Code

While several researchers had hypothesized that solving Parsons problems would result in more efficient learning than writing the equivalent code, this between-subjects study was the first study that tested this hypothesis. It also compared solving Parsons

problems with paired distractors to fixing code with the same errors as the distractors, since both types of problems have the advantage that the learner doesn't have to type the code. Students in the Parsons problem condition completed the four practice problems in significantly less time than those in the fix or write code conditions and had comparable learning gains from the pretest to the immediate posttest and the delayed posttest one week later (B. J. Ericson et al., 2017). There was no difference in the learning gains by condition, which implies that students learned just as much from solving Parsons problem as from fixing or writing code. However, the learning gains may not have been solely due to learners completing the practice problems, since there were other materials in the study that the students could have learned from such as a review of lists and the answers to the practice problems. Also, it is possible that students merely performed better on the immediate posttest and the delayed posttest due to repeated exposure to the same or similar problems.

7.3 Summary of Testing the Effectiveness of Intra-Problem Adaptation

The observational study of 11 teachers solving eight intra-problem adaptive Parsons problems and eight non-adaptive Parsons problems allowed us to examine if learners understood the adaptation process, the effectiveness of the adaptation with respect to getting the problem correct, teachers' preference for adaptive or non-adaptive Parsons problems, and the perception of the usefulness of solving Parsons problems in learning to fix and write code.

While learners understood the intra-problem adaption process when it disabled distractors that were used in the solution and combined blocks, leaners were confused

when it disabled distractors that had not been used in the solution (i.e. were still in the source area on the left) or provided the indentation. After this study, we stopped disabling distractors that hadn't been used in the solution. While two teachers solved the problem after indentation was provided, others didn't realize that the indentation indicated which blocks needed to be moved outside of the loop. Further research needs to be done to determine if the intra-problem adaptation process should provide indentation or not.

The adaptation process appeared to be effective in that all the teachers solved all the adaptive problems in the study and two teachers did not solve one of the non-adaptive problems. However, teachers had to be encouraged to use the adaptation at times and might have given up on solving the problem without that encouragement. A log file analysis of students solving the same 16 problems showed that not all of the students who used the adaptation solved the problems. However, a significantly higher percentage of students who used the adaptation got the problem correct than the students who didn't use the adaptation, thus providing additional evidence that using the adaption correlates with a higher percentage of learners getting the problem correct. However, this is likely due to the percentage of learners who give up on solving the problem before help is enabled.

All but one teacher preferred the adaptive Parsons problems to the non-adaptive ones. The other teacher suggested providing audio help or showing a similar example instead of dynamically making the problem easier to solve. The teachers liked that the students couldn't get help (adaptation) until they had made at least three full attempts to solve the problem.

The three undergraduate students that we tested our materials on as well as the 11 teachers all thought that solving Parsons problems with distractors helped them learn to identify common syntax problems. They also felt that Parsons problems helped them learn to fix code with similar syntax errors and write code from scratch. However, this perception should be empirically tested.

7.4 Summary of Adaptive Parsons versus Parsons versus Write Code with a Control Group Solving Off-Task Adaptive Parsons Problems

This between-subjects study was very similar to the study that compared solving non-adaptive Parsons problems to fixing code and writing code. However, this time we compared solving both intra-problem and inter-problem adaptive Parsons problems to solving non-adaptive Parsons problems and writing code. Intra-problem adaption makes the current problem easier by removing distractors, providing indentation, and combining blocks. Inter-problem adaptation changes the next Parsons problem to be either easier or harder depending on the number of attempts it took the learner to solve the current problem. The next problem can be made harder by adding more distractors or un-pairing distractors. The next problem can be made easier by removing distractors or pairing them with the correct code blocks.

We addressed some of the weaknesses from the previous between-subjects study by including a control group that solved off-task adaptive Parsons problems on turtle graphics. We also removed the review material and didn't show the students the answers after they finished each practice problem. These changes were intended to strengthen the case that any learning gains were due to solving the instructional practice problems.

Both the adaptive and non-adaptive on-task Parsons problem groups completed the four practice problems significantly faster than the write code group. There was also a statistically significant difference in the scores from pretest to posttest, which provides evidence of learning gains. However, there was no statistical significance between the control group and the non-adaptive Parsons group or the write code group. There was a significant difference between the adaptive Parsons problem group and the control group which provides initial evidence that solving adaptive Parsons problems can lead to a learning gain. However, since the students in the control condition who solved off-task problems showed the same learning gains from pretest to posttest as those in the write code and non-adaptive Parsons conditions, this implies that at least some of the learning gains could be from repeated exposure to the same or similar problems, rather than solely from solving the instructional practice problems. Further research needs to be done to compare the learning gains from solving adaptive Parsons problems, and non-adaptive Parsons problems, versus writing the equivalent code.

7.5 Short Term Future Work

The research studies described in this thesis have left several open questions about Parsons problems that I want to answer in the short term. I want to empirically test that solving Parsons problems with distractors improves performance on fix and write code problems, test a change to the intra-problem adaptation process, and test if numbered labels help learners realize that the distractor and correct blocks are shown paired. In addition, I want to improve the Parsons problem software to allow it to generate distractors based on the user's performance and allow constraint-based Parsons problems.

In constraint-based Parsons problems it would be possible to have more than one correct solution, since the solution would have to satisfy a set of constraints rather than a strict ordering of the blocks.

7.5.1 Testing learning gains from solving adaptive and non-adaptive Parsons problems

The observational study of learners solving both adaptive and non-adaptive Parsons problems provided evidence that undergraduate students and teachers both felt that solving Parsons problems with distractors helped them learn to fix and write code. However, that perception needs to be empirically tested. The two between subject studies both had statistically significant gains from pretest to posttest for all conditions, however we can't be sure that these learning gains were from the practice condition rather than from repeated exposure to the same or similar problems. Indeed, since the control group in the second between-subjects study had the same learning gains from pretest to posttest as the write code and non-adaptive Parsons groups, it looks like the learning gains were not solely due to the instructional practice.

I would like to do a multi-institutional study to test the efficiency and effectiveness of solving Parsons problems versus writing code. A multi-institutional study would provide a large number of subjects for greater statistical power. I would like to include a variety of types of institutions in the study, not just research-intensive institutions, to test Parsons problems on a larger range of subjects. It would also be interesting to gather the subject's SAT reading score to see if there is any correlation with performance. To reduce the effect from repeated exposure to the same or similar

problems, I would make the immediate posttest isomorphic to the pretest and change the order of the problems as well.

7.5.2 Testing a change to the intra-problem adaptation process

While two teachers were able to solve an intra-problem adaptive Parsons problem after indentation was provided, others were confused by it and failed to use the implicit clues that it provided. I want to modify the intra-problem adaptation to only remove distractor blocks and combine blocks (i.e. no longer provide indentation). One way to test this change is via AB testing with two versions of the student CSP ebook. One version would continue to provide the indentation as part of the intra-problem adaptation process and the other would remove it. A log file analysis of each version would indicate the importance of providing the indentation.

7.5.3 Testing numbered labels to indicate the pairing of a distractor and correct code block

Several teachers didn't notice the purple edge decorations which were intended to visual signify that the distractor and correct blocks were paired (shown together) and that the learner only needed to pick one of the two blocks to use in the solution. In a previous study, only a few students used both the correct and distractor blocks in a solution when we provided the purple edge decorations and also used the same subgoal labels as comments on both blocks. This implied that the subgoal labels helped the students realize that the distractor and correct blocks were paired. However, I do not want to rely on the author to create the same subgoal label on both the correct and distractor code blocks.

We modified the software to include an option to add numbered labels (like 1a and 1b for a distractor and correct pair) on the right side of the code block. The labels will serve two purposes. One purpose is to allow users to refer to blocks by their labels when discussing them, which will make it easier to allow groups of users to work together on Parsons problems. The second purpose is to add an additional visual signifier that the blocks are paired since the labels will match. One advantage of this approach is that the labels will remain on the blocks even if they separated, such as when one block has been used in the solution area and the other is still in the source area. I would like to do an observational study of learners working through Parsons problems with paired distractors with labels to test if the learners understand what the labels are implying. I would also like to add the labels to the Parsons problems in the student version of the ebook to test their effect by seeing how many people use both a correct block and a distractor block in the same solution.

7.5.4 Improvements to the Parsons problem software

The current Parsons problem software that we have been using requires the distractors to be specified when the problem is authored. I want to explore using rules to modify the correct code to generate distractors as needed. Ideally, the distractors would be based on what the learner is currently struggling with, such as using the incorrect case. Distractors could also be based on common errors or misconceptions.

One of the limitations of the current Parsons problem software is that there can only be one correct solution. I would like to explore using constraints to specify that a block needs to be before another or after another instead of specifying a strict order. This

would be helpful when the order doesn't really matter for a set of blocks. For example, when you declare several variables, the order of the declarations usually doesn't matter.

7.6 Long Term Future Work

There are many things that I want to accomplish over the next five to ten years. In particular, I would like to encourage the use of Parsons problems, incorporate Artificial Intelligence, and add support for group work.

7.6.1 Encourage others to use Parsons problems

I would like to encourage more use of Parsons problems as a type of practice problem for learning how to program. In order to accomplish this, it must be easier for others to reuse the hundreds of Parsons problems that I and the students who work for me have created. Currently, other researchers can create a custom course from one of the existing ebooks on Runestone Interactive or get the source for the ebooks on github and create their own ebooks. A research group in Finland has translated the teacher AP CSP ebook into Finnish and others have been translating it to Chinese and Spanish. Researchers at the University of Pittsburgh have also asked to incorporate the Java Parsons problems from my ebook for the AP CSA course in their work, but they wanted to pass in the user name and get back the results, which currently isn't supported. I would like to modify the Runestone Interactive server to make it easier for others to reuse all the material in the ebooks, including the Parsons problems and multiple-choice questions. In addition, I would like to make the usage data accessible, including the percentage of people who get the problem right on the first attempt, how many attempts it takes for

75% of the people to get the problem correct, and how many people give up and never solve the problem. This data would be useful in determining good problems for peer instruction. A good peer instruction question is one that 40-60% of the students get right on their first attempt (Kober, 2015). Peer instruction was originally developed by Eric Mazur of Harvard to improve learning outcomes in physics. In peer instruction, the instructor presents some material or students work through material outside of class, then the instructor displays a multiple-choice question that contains distractors based on common misconceptions. The students have a set amount of time to answer the question individually. The students answer the question again, but after discussing their answer with another student (a peer). Finally, the instructor reveals the correct answer, and addresses misconceptions and takes questions from the students (Crouch & Mazur, 2001). Evaluation of peer instruction has shown increased student understanding and engagement in many fields including physics (Crouch & Mazur, 2001), biology (Knight & Wood, 2005) and computer science (Porter et al., 2016) (Porter, Lee, Simon, & Zingaro, 2011). I would also like to test using Parsons problems in peer instruction questions.

7.6.2 Incorporate Artificial Intelligence

I have been interested in Artificial Intelligence for many years and have done work in case-based reasoning. I want to incorporate AI in my future research. One goal is to use machine learning to predict students who are at risk of failing a programming course based on their ebook interaction data. Other researchers have used peer instruction (clicker) data to try to predict students who are at-risk (Liao, Zingaro, Laurenzano,

Griswold, & Porter, 2016; Porter, Zingaro, & Lister, 2014), but I think the ebook interaction data would provide a richer picture of student performance. I would also like to test scalable ways to provide at-risk students with additional help. One of my current research projects, Rise Up 4 CS, has been providing both remote and in-person extra help sessions to underrepresented high school students to help them succeed in their Advanced Placement Computer Science courses, both A and Principles (B. Ericson, Engelman, McKlin, & Taylor, 2014; B. Ericson & McKlin, 2015; B. J. Ericson, Parker, & Engelman, 2016). The help sessions are led by undergraduate students who also serve as near peer role models. The undergraduate students use the interactive ebooks that we have been creating for both the AP CSA and AP CSP courses during the help sessions.

I would also like to allow the user to ask the ebook questions and get answers like in the *Inquiry Biology* textbook (Chaudhri et al., 2013). This interactive ebook used artificial intelligence to help students understand the large number of concepts and relationships in an introductory biology course. Students could ask free-form questions and the ebook would display a list of similar questions that it could answer. Students could highlight text and the ebook would suggest questions related to that material. It would also suggest follow-up or more in-depth questions. Community college students who used this interactive ebook had 10% higher grades on homework and posttest problems than the students who used the paper textbook or just a digital version of the paper textbook. One way to create this type of interactive ebook might be to use technology, like IBM's Watson. This technology was used to create the first version of a

virtual teaching assistant named Jill Watson, who answered routine questions in an online forum (Goel & Polepeddi, 2018).

7.6.3 Add support for group work

The current interactive ebook platform allows the instructor to create assignments that require students to attempt particular problems in an ebook, but it assumes that students are working independently. Another active learning technique, peer-led team learning, has also increased student engagement and retention (Horwitz et al., 2009). In peer-led team learning students work in small groups on challenging problems with the help of an undergraduate student leader who has previously excelled at the course. The undergraduate student leader facilitates the group and ensures that all of the student participate and understand the concepts. A study of peer-led team learning with underrepresented students in computer science found that it improved student engagement and retention (Horwitz et al., 2009). I would like to provide support for peer-led team learning in the interactive ebooks.

I think interactive electronic books and other forms of online learning are in their infancy and have a great potential to improve education. As Soloway, Guzdial, and Hay suggested, the power of computers should be used to help humans realize their full potential (Soloway et al., 1994). I want to improve online learning to help achieve that goal.

APPENDIX A. STUDY MATERIALS

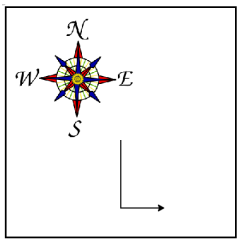
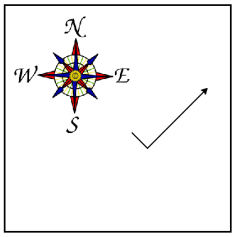
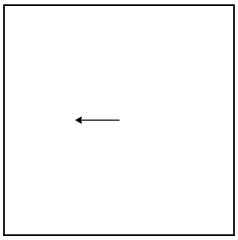
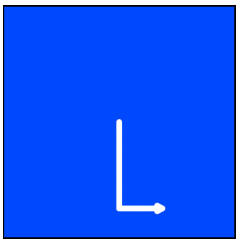
A.1 Observational Study and Log File Analysis from Chapter 3

I added eleven Parsons problems to chapter four of the *How to Think Like a Computer Scientist* interactive ebook before the teacher observation study described in chapter 3. See https://runestone.academy/runestone/static/thinkcspy/index.html for the online version of this ebook. See https://github.com/RunestoneInteractive/thinkcspy for the source for this ebook. You need to install Runestone to convert the source to html. See http://runestoneinteractive.org/instructors.html for directions on how to set up Runestone and build this ebook from the source.

Teachers worked through the first three chapters of this ebook on their own and then were observed remotely as they worked through the fourth chapter which included turtle graphics and the Parsons problems. The log file described in chapter 3 was from university and high school students use of this ebook.

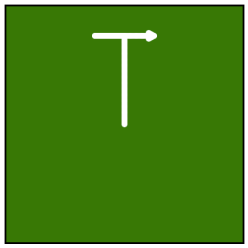
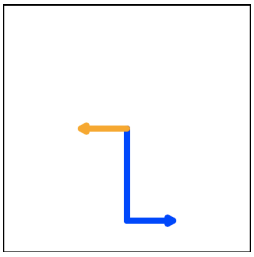
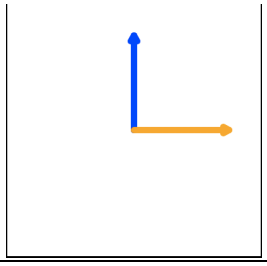
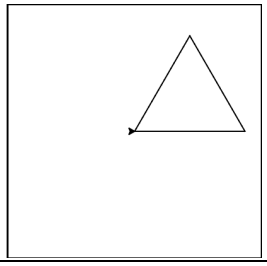
The following table shows pictures, a short description, and the starting mixed-up code for the 11 Parsons problems in chapter four of this ebook. Learners had to drag the blocks into the correct order with the correct indentation. They clicked a *Check Me* button to get feedback on their solution. Blocks that needed to be moved were shown in red. Blocks that were indented incorrectly had red highlights on the left edge of the block.

Table 27. The 11 Parsons problems in chapter four of the How to Think Like a Computer Scientist Ebook

Problem 1 Draw L	**Problem 2** Draw check	**Problem 3** Draw line	**Problem 4** Draw white L on blue
			

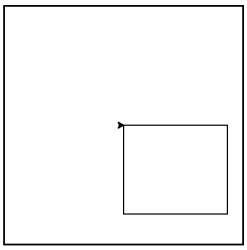
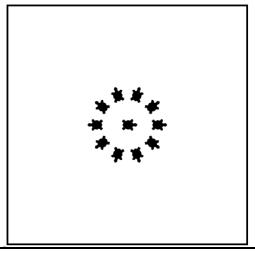
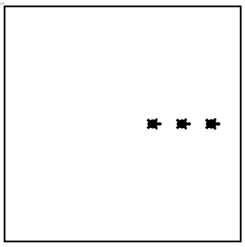
Drag from here
`ella.left(90)`
`ella.forward(75)`
 	
`import turtle`
`window = turtle.Screen()`
`ella = turtle.Turtle()`
 	
`ella.right(90)`
`ella.forward(150)`
	Drag from here
	`maria.left(90)`
	`maria.forward(150)`
	
	`window = turtle.Screen()`
	
	`import turtle`
	
	`maria.right(45)`
	`maria.forward(75)`
	
	`maria = turtle.Turtle()`
		Drag from here
		`jamal = turtle.Turtle()`
		
		`window = turtle.Screen()`
		
		`jamal.forward(75)`
		
		`import turtle`
		
		`jamal.left(180)`
			Drag from here
			`jamal.right(90)`
			`jamal.forward(150)`
			
			`jamal.color("white")`
			`jamal.pensize(10)`
			
			`wn.bgcolor("blue")`
			`jamal = turtle.Turtle()`
			
			`import turtle`
			`wn = turtle.Screen()`
			
			`jamal.left(90)`
			`jamal.forward(75)`
			`wn.exitonclick()`

Table 27. Continued

Problem 5 Draw white T on green	**Problem 6** Draw L in Blue and orange line to the west	**Problem 7** Draw blue line north and orange line east	**Problem 8** Draw a triangle
			

Drag from here
`jamal.left(90)`
`jamal.forward(50)`
`jamal.left(90)`
`jamal.forward(150)`
`wn.exitonclick()`
`jamal.right(180)`
`jamal.forward(100)`
`import turtle`
`wn = turtle.Screen()`
`wn.bgcolor("green")`
`jamal = turtle.Turtle()`
`jamal.color("white")`
`jamal.pensize(10)`
	Drag from here
	`tina = turtle.Turtle()`
	`tina.pensize(10)`
	`tina.color("orange")`
	`tina.left(180)`
	`tina.forward(75)`
	`jamal = turtle.Turtle()`
	`jamal.pensize(10)`
	`jamal.color("blue")`
	`jamal.right(90)`
	`jamal.forward(150)`
	`jamal.left(90)`
	`jamal.forward(75)`
	`import turtle`
	`wn = turtle.Screen()`
	`wn.exitonclick()`
		Drag from here
		`tina = turtle.Turtle()`
		`tina.pensize(10)`
		`tina.color("orange")`
		`tina.forward(150)`
		`jamal.left(90)`
		`jamal.forward(150)`
		`wn = turtle.Screen()`
		`import turtle`
		`wn.exitonclick()`
		`jamal = turtle.Turtle()`
		`jamal.color("blue")`
		`jamal.pensize(10)`
			Drag from here
			`wn.exitonclick()`
			`marie.left(120)`
			`import turtle`
			`wn = turtle.Screen()`
			`marie = turtle.Turtle()`
			`marie.forward(175)`
			`# repeat 3 times`
			`for i in [0,1,2]:`

Table 27. Continued

Problem 9 Draw rectangle	**Problem 10** Stamp Circle	**Problem 11** Stamp Line
		

Drag from here
`carlos.forward(175)`
`wn.exitonclick()`
`# repeat 2 times`
`for i in [1,2]:`
`carlos.forward(150)`
`carlos.right(90)`
`carlos.right(90)`
`import turtle`
`wn = turtle.Screen()`
`carlos = turtle.Turtle()`
	Drag from here
	`jose.stamp()`
	`import turtle`
	`wn = turtle.Screen()`
	`jose = turtle.Turtle()`
	`jose.shape("turtle")`
	`jose.penup()`
	`jose.forward(-50)`
	`jose.forward(50)`
	`jose.right(36)`
	`wn.exitonclick()`
	`for size in range(10):`
		Drag from here
		`nikea.forward(50)`
		`import turtle`
		`wn = turtle.Screen()`
		`nikea.stamp()`
		`nikea.shape("turtle")`
		`for size in range(3):`
		`nikea = turtle.Turtle()`
		`wn.exitonclick()`
		`nikea.penup()`

A.2 Parsons versus Fix and Write Study from Chapter 4

Online materials

The online materials include the familiarization and practice questions, pretest, review of lists and ranges, instructional worked examples plus practice pairs, the immediate posttest, and the delayed posttest which was given one week later.

The online materials are at the following URLs, but you must be a registered user to access them. Registration is free.

- http://tinyurl.com/Ex1Parsons
- http://tinyurl.com/Ex1Fix
- http://tinyurl.com/Ex1Write
- http://tinyurl.com/Ex1Post
- http://tinyurl.com/Ex1SecondPost

The source for the online materials is all on github as:

- https://github.com/ericsonga/ex1Parsons
- https://github.com/ericsonga/ex1Fix
- https://github.com/ericsonga/ex1Write
- https://github.com/ericsonga/ex1Post
- https://github.com/ericsonga/Ex1SecondPost

Experiment Procedure

The experiment procedure shown below was projected in the closed lab. Students were also given a hardcopy of it after they signed the consent form. Students selected a piece of paper from a plastic bag that contained the URL of the online materials for their experimental condition.

Experiment Procedure

Please do not talk to others during the Experiment
You may leave to use the restroom, but leave all experiment materials in this room
If you have a question, please raise your hand or come down to the front of the room

1) You should have a consent form and two pieces of scratch paper – all with the same 6-digit code on them (one marked pre and one marked post). Fill out the consent form (assent if under 18). Take a picture of the 6-digit code with your cell phone and turn in the consent form at the front of the room and pick a URL. Put your cell phone away.
2) Open a browser window (Chrome, Firefox, or Safari) and make it full screen.

3) Create a login using the 6-digit code as your username at http://tinyurl.com/Ex1Reg. Use a course name of *thinkcspy*.
4) Fill out the demographic survey at http://tinyurl.com/Ex1-GT-D
5) Go to the URL you got when you turned in your consent form. Work through the following material in order (starting with the first link in the table of contents) and go to the next page each time you are finished with the current page. **Use the scratch paper with the word *pre* on it if needed.** On timed exams, be sure to click ***"Start"*** to start the timed exam, ***"Run"*** or ***"Check Me"*** to check your answers (if allowed), and ***"Finish Exam"*** to end a timed exam. You can optionally click the ***"Mark as completed"*** button at the bottom of the page. Use the **right arrow**  at the bottom right of the page to go to the next page. If you don't see the right arrow make sure that you clicked the **"Finish Exam"** button. **Please do not use the back or forward button on the browser or click on the table of contents link at the top left!**
    a. Practice material
        2 multiple-choice questions, 1 fix code problem, 1 order code problem and 1 write code problem
    b. Pretest material
        5 multiple-choice questions, 1 fix code problem, 1 order code problem, and 1 write code problem
    c. List Review (basics)
    d. Instructional materials
        4 example and practice pairs on the same page
6) **Turn in the scratch paper that has the word *pre* on it at the front of the room.**
7) Fill out the cognitive load survey at http://tinyurl.com/Ex1CogLoad Answer the questions about the difficulty of the example plus practice section.
8) Do the post-test at http://tinyurl.com/Ex1PostGT **using the scratch paper that has the word *post* on it if needed.**
9) Turn in your pen/pencil and the scratch paper that has the word *post* on it as you leave.

Student Demographic Survey

Demographics and Prior Programming Experience

1. Please enter the 6 digit code that was assigned to you for this experiment. If you do not know it please contact Barbara Ericson at ericson@cc.gatech.edu.

2. What is your name (first name followed by last name)?

3. What is your age?

- under 16
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23-29
- 30-39
- 40-49
- 50-59
- 60 or older

4. What gender do you identify with?

- Male
- Female
- Other

5. What is your race? (check all that apply)

- [ ] American Indian or Alaskan Native
- [ ] Asian
- [ ] Black or African American
- [ ] Hispanic/Latino/Latina
- [ ] Pacific Islander
- [ ] White
- [ ] Other (please specify)

6. What was your first spoken language?

- ( ) Arabic
- ( ) English
- ( ) French
- ( ) Hindi
- ( ) Mandarin
- ( ) Spanish
- ( ) Other (please specify)

7. How comfortable are you with reading English?

- ( ) Very uncomfortable
- ( ) Uncomfortable
- ( ) Comfortable
- ( ) Very comfortable

8. What was your high school grade point average (weighted) on a 4.0 scale?

- ◯ Over 4.0
- ◯ over 3.5 to 4.0
- ◯ over 3.0 to 3.5
- ◯ over 2.5 to 3.0
- ◯ over 2.0 to 2.5
- ◯ over 1.5 to 2.0
- ◯ over 1.0 to 1.5

9. What is your current (college) overall grade point average? Use 0 if this is your first semester of college.

10. What is your current major?

- ◯ Computer Science
- ◯ Computational Media
- ◯ Computer Engineering
- ◯ Electrical Engineering
- ◯ Other (please specify)

11. Please rate the following

	No ability	Some ability	About average ability	Better than average ability	Expert
Your ability to read and understand a Python program	◯	◯	◯	◯	◯
Your ability to fix errors (debug) in a Python program	◯	◯	◯	◯	◯
Your ability to write a working Python program	◯	◯	◯	◯	◯

12. What grade do you expect to get in your programming course (1301)?

- ◯ A
- ◯ B
- ◯ C
- ◯ D
- ◯ F

13. Is this course your first experience with programming?

- ◯ Yes
- ◯ No

Prior Programming Experience

14. What other programming courses have you taken and where (college or high school)?

15. How long have you been programming?

- ◯ Less than 1 year
- ◯ 1-2 years
- ◯ 3-5 years
- ◯ 6-10 years
- ◯ Over 10 years

Thank You!

Thank you for filling out the demographic survey! Your answers have been recorded.

A.3 Observational Study of Teachers Solving Adaptive and Non-adaptive Parsons problems

Teachers were observed solving 16 Parsons problems in chapter five of an ebook for the Advanced Placement Computer Science Principles course. Half of the Parsons problems were intra-problem adaptive which means that the problem could be made easier if the user asked for help and half were not. Half of the Parsons problems had paired distractors and half had distractors randomly mixed in with the correct code.

Table 28. The 16 Parsons problems in the observational study and log file analysis

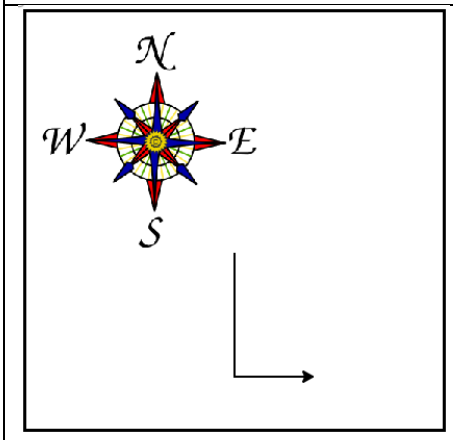
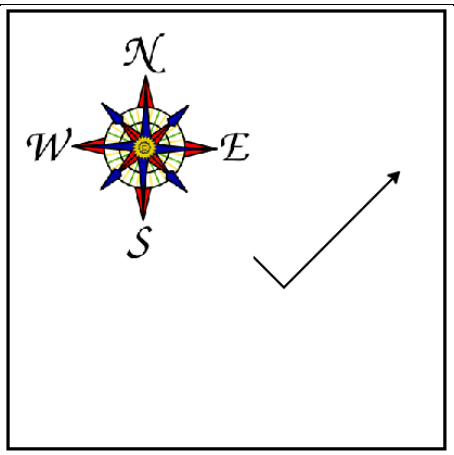
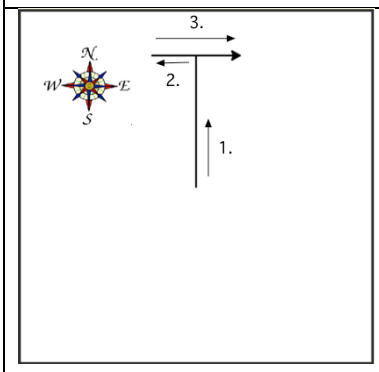
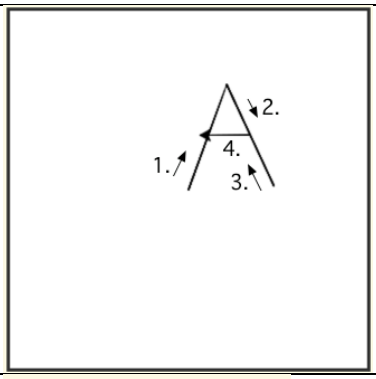
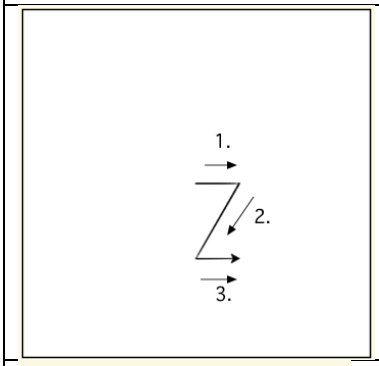
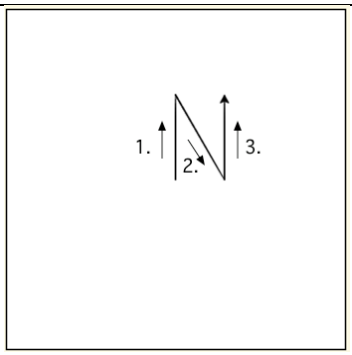
Problem 1 Draw a L	**Problem 2** Draw a check
	
Drag from here ella.left(90) ella.turn(90) ella.right(90) ella.go(75) ella.forward(75) ella = Turtle() from turtle Import * from turtle import * ella.forward(150) space = Screen() space = screen()	Drag from here maria.Forward(75) maria.forward(75) maria.right(45) maria.left(45) maria = Turtle maria = Turtle() from turtle import * space = Screen() maria.left(90) maria.right(90) maria.forward(150)

Table 28. Continued

Problem 3 Draw a T	**Problem 4** Draw A
	

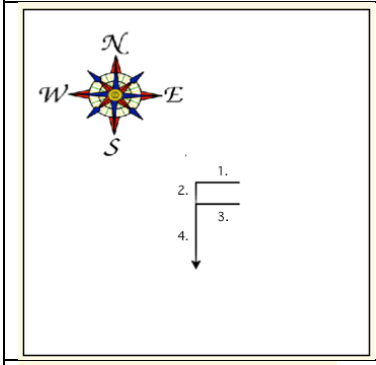
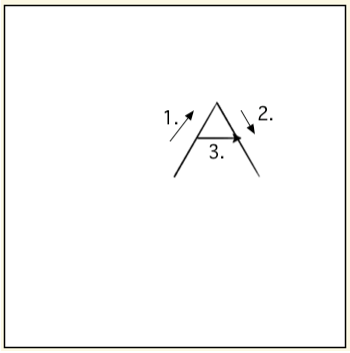
Drag from here
`jamal.turn(180)`
`jamal.Left(90)`
`jamal.Forward(150)`
`from turtle import *`
`jamal.forward(100`
`jamal.forward(100)`
`jamal.right(180)`
`jamal.forward(150)`
`jamal.left(90)`
`jamal.forward(50)`
`space = Screen()`
`jamal = Turtle()`
`jamal.left(90)`
	Drag from here
	`from turtle import *`
	`space = Screen()`
	`jamal.forward(100)`
	`jamal.forward(100)`
	`jamal.right(135)`
	`jamal.forward(100`
	`jamal.right(180)`
	`jamal.forward(50)`
	`jamal.left(70)`
	`jamal.right(180)`
	`jamal.Forward(50)`
	`jamal = Turtle)`
	`jamal.left(20)`
	`jamal = Turtle()`
	`jamal.left(65)`
	`jamal.forward(45)`

Table 28. Continued

Problem 5 Draw Z	**Problem 6** Draw N
	

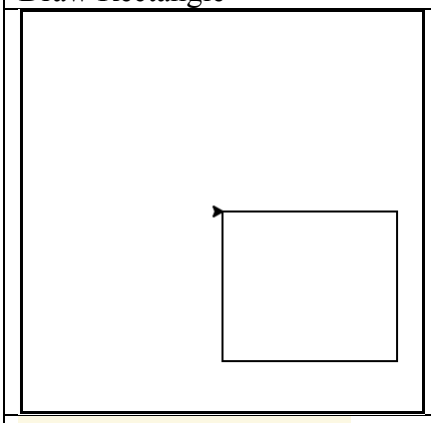
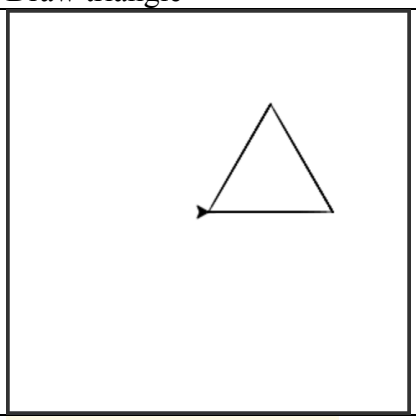
Drag from here
`alex.forward(50)`
`alex.turn(120)`
`alex.forward(50)`
`alex.right(120)`
`alex.forward(100)`
`alex.forward(50)`
`alex = turtle()`
`alex = Turtle()`
`from turtle Import *`
`from turtle import *`
`space = Screen()`
`space = screen()`
`alex.left(120)`
	Drag from here
	`from turtle import *`
	`ella.forward(100)`
	`ella.Forward(100)`
	`ella = Turtle()`
	`ella = Turtle`
	`ella.left(150)`
	`ella.left(90)`
	`ella.forward(100)`
	`ella.right(90)`
	`ella.forward(100)`
	`ella.left(150)`
	`ella.forward(116)`
	`ella.right(150)`
	`ella.forward(116)`
	`space = Screen()`

Table 28. Continued

Problem 7 Draw F	**Problem 8** Draw A (with penup)
	

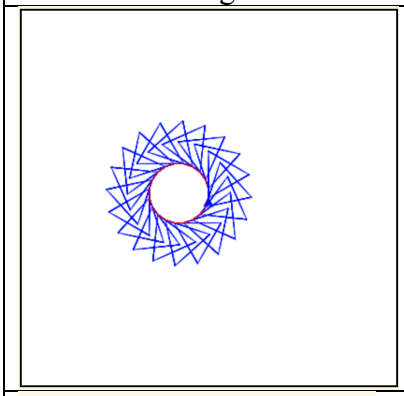
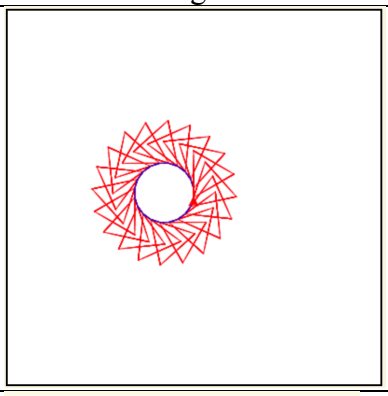
Drag from here
`anu.penDown()`
`anu.goTo(0,60)`
`from turtle import *`
`space = Screen()`
`anu = Turtle()`
`anu.goto(0,60)`
`anu.penup()`
`anu.right(90)`
`anu.forward(50)`
`anu.pendown()`
`anu.left(90)`
`anu.forward(50)`
`anu.left(90)`
`anu.forward(100)`
`anu.forward(50)`
`anu.penUp()`
	Drag from here
	`from turtle import *`
	`ella = Turtle()`
	`ella = Turtle`
	`ella.left(60)`
	`ella.forward()`
	`ella.right(120)`
	`ella.forward(100)`
	`space = Screen()`
	`ella.Left(60)`
	`ella.forward(40)`
	`ella.penup()`
	`ella.goto(30,50)`
	`ella.pendown()`
	`space = screen()`
	`ella.left(60)`
	`ella.forward(100)`
	`ella.left(60)`
	`ella.forward(40)`

Table 28. Continued

Problem 9 Draw Rectangle	**Problem 10** Draw triangle
	

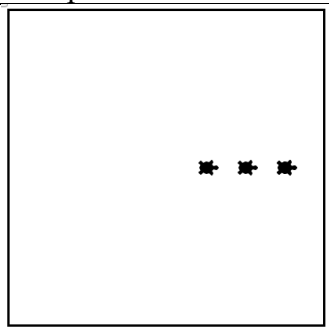
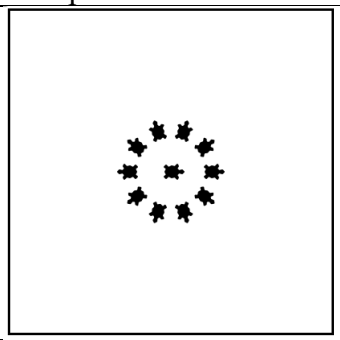
Drag from here
`from turtle import *`
`from Turtle import *`
`carlos.forward(150)`
`carlos.right(90)`
`carlos.forward(150)`
`carlos.turn(90)`
`space = Screen()`
`carlos = Turtle()`
`# repeat 2 times`
`for i in [1,2]`
`# repeat 2 times`
`for i in [1,2]:`
`carlos.Forward(175)`
`carlos.forward(175)`
`carlos.right(90)`
	Drag from here
	`# repeat 3 times`
	`for i in range(3):`
	`# repeat 3 times`
	`for i in range(3)`
	`marie = Turtle()`
	`marie.left(120)`
	`marie.turn(120)`
	`marie.forward(100)`
	`marie.forward(100`
	`space = screen()`
	`space = Screen()`
	`from turtle import *`

Table 28. Continued

Problem 11 Draw blue triangles and red circle	**Problem 12** Draw red triangles and blue circle
	

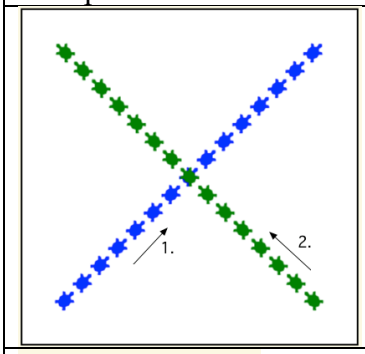
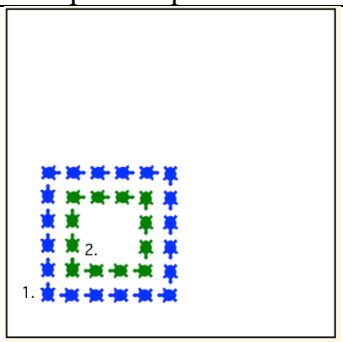
Drag from here
`mateo.color("blue")`
`mateo.forward(50)`
`mateo.right(120)`
`for sides in range(4):`
`mateo.color("red")`
`mateo.forward(10)`
`mateo.left(12)`
`for sides in range(3):`
`mateo.color("red")`
`mateo.forward(10)`
`mateo.left(18)`
`for repeats in range(20):`
`for repeats in range(20)`
`from turtle import *`
`from sys import *`
`setExecutionLimit(50000)`
`wn = Screen()`
`mateo = Turtle()`
`mateo.setheading(90)`
	Drag from here
	`for sides in range(3):`
	`wn = Screen()`
	`mateo = Turtle()`
	`mateo.setheading(90)`
	`for sides in range(3)`
	`mateo.color("blue")`
	`mateo.forward(10)`
	`mateo.left(18)`
	`for repeats in range(20):`
	`mateo.color("red")`
	`mateo.forward(50)`
	`mateo.right(120)`
	`from turtle import *`
	`from sys import *`
	`setExecutionLimit(50000)`
	`mateo.color("red")`
	`mateo.forward(50)`
	`mateo.right(60)`
	`for repeats in range(20)`

Table 28. Continued

Problem 13 Stamp three turtles in a line	**Problem 14** Stamp a circle of turtles
	

Drag from here
`from turtle import *`
`space = Screen()`
`space.setup(400,400)`
`nikea.stamp`
`nikea.stamp()`
`for size in range(3):`
`for size in range(3)`
`nikea = Turtle()`
`nikea.forward(50)`
`nikea.penup()`
`nikea.penUp()`
`nikea.shape(turtle)`
`nikea.shape("turtle")`
	Drag from here
	`for size in range(10)`
	`for size in range(10):`
	`from turtle import *`
	`space = Screen()`
	`jose = Turtle()`
	`jose.shape("turtle")`
	`jose.penup`
	`jose.shape("turtle")`
	`jose.penup()`
	`jose.forward(50)`
	`jose.right(20)`
	`jose.right(36)`
	`jose.stamp()`
	`jose.Stamp()`
	`jose.forward(-50)`
	`jose.forward(-25)`

Table 28. Continued

Problem 15 Stamp an X with turtles	**Problem 16** Stamp two squares
	

Drag from here
`space = Screen()`
`nick.color("blue")`
`for num in range(15):`
`  nick.stamp()`
`  nick.forward(30)`
`nick.stamp()`
`nick.forward(30)`
`from turtle import *`
`nick.goto(150,-150)`
`nick.left(45)`
`nick = Turtle()`
`nick.shape("turtle")`
`nick.goto(-150,-150)`
`nick.left(45)`
`nick.color("green")`
`for num in range(14):`
`nick.penUp()`
`space = screen()`
`nick.goto(-150,-150)`
`nick.left(90)`
`nick.goto(150,-150)`
`nick.left(90)`
`nick.penup()`
	Drag from here
	`nick.goto(-150,-150)`
	`nick.left(90)`
	`for num in range(3):`
	`nick = Turtle()`
	`nick.shape("Turtle")`
	`nick = Turtle()`
	`nick.shape("turtle")`
	`for num in range(2):`
	`nick.left(90)`
	`nick.color("blue")`
	`for count in range(4):`
	`  for num in range(5):`
	`  nick.stamp()`
	`  nick.forward(30)`
	`  nick.right(90)`
	`nick.penup()`
	`nick.stamp()`
	`nick.forward(30)`
	`from turtle import *`
	`space = Screen()`
	`nick.right(90)`
	`nick.goto(-120,-120)`
	`nick.color("green")`
	`for count in range(4):`
	`nick.goto(-150,-150)`
	`nick.right(90)`

Teacher Survey

Prior Programming Experience

Thank you for expressing an interest in our research study for high school teachers with less than 3 months experience with textual programming. Please fill out the following questions to verify that you qualify for the study and to help us understand your prior programming experience.

* 1. Please enter the following:

Name (first and last:	
E-mail address:	
State:	
How many years have you been teaching (any type of course)?	
How many years have you taught computing courses?	
What is your certification / endorsement / license (like business, math, or computer science)?	
How old are you?	

2. What gender do you identify as?

- ○ Male
- ○ Female

3. What race do you identify as (pick all that apply)?

- [ ] African American / Black
- [ ] Asian
- [ ] Caucasian / White
- [ ] Hispanic
- [ ] Native American
- [ ] Alaskan Native
- [ ] Pacific Islander
- [ ] Other (please specify)

* 4. Do you have less than 3 months experience programming in a textual programming language like Python, C,C++, C# Java, etc?

- ( ) Yes
- ( ) No

Prior Programming Experience

* 5. Which of the following environments or programming languages have you used?

- [ ] Scratch
- [ ] Snap
- [ ] Alice
- [ ] App Inventor
- [ ] Python
- [ ] Java
- [ ] C
- [ ] C++
- [ ] C#
- [ ] Lisp
- [ ] Smalltalk
- [ ] Squeak
- [ ] Java Script
- [ ] LEGO NXT-G
- [ ] Kodu
- [ ] GameMaker
- [ ] GameSalad

Other (please specify)

* 6. Please pick the answer that best describes your programming history.

- ○ I have no experience with programming
- ○ I did some programming years ago, but nothing lately
- ○ I have less than one year experience with drag-and-drop environments like Scratch, Snap, Alice, or App Inventor
- ○ I have lots of experience with drag-and-drop environments like Scratch, Snap, Alice, or App Inventor

* 7. Which of the following concepts have you used when programming?

	Have not heard of this	Have heard of this, but never used it in a program	Used in a program	Expert
Variables	○	○	○	○
Iteration - Loops (for loop, for each loop, repeat, forever, while)	○	○	○	○
Conditionals - If or If/else	○	○	○	○
Arrays	○	○	○	○
Lists	○	○	○	○
Inheritance	○	○	○	○
Recursion	○	○	○	○

8. Which computing courses have you taught?

- ☐ None
- ☐ Exploring Computer Science (ECS)
- ☐ Introduction to Digital Technology
- ☐ Beginning Programming
- ☐ Intermediate Programming
- ☐ Web Development
- ☐ Advanced Placement Computer Science Principles
- ☐ Advanced Placement Computer Science A
- ☐ Games, Apps, and Society
- ☐ Other (please specify)

A.4 Adaptive Parsons versus Parsons versus Write Code from Chapter 6

Online materials

The online materials include the familiarization and practice questions, pretest, instructional worked examples plus practice pairs, the immediate posttest, and the delayed posttest which was given one week later.

The online materials are at the following URLs, but you must be a registered user to access them. Registration is free.

- **http://tinyurl.com/Ex3ParsonsAdapt**
- **http://tinyurl.com/Ex3Parsons**
- **http://tinyurl.com/Ex3Write**
- **http://tinyurl.com/Ex3Turtle**
- **http://tinyurl.com/Ex3SecondPost**

The source for the online materials is all on github as:

- https://github.com/ericsonga/ex3ParsonsAdapt
- https://github.com/ericsonga/ex3Parsons
- https://github.com/ericsonga/ex3Write
- https://github.com/ericsonga/ex3Turtle (control group)
- https://github.com/ericsonga/Ex3SecondPost

Experiment Procedure

The experiment procedure shown below was projected in the closed lab. Students were also given a hardcopy of it after they signed the consent form. Students selected a piece of paper from a plastic bag that contained the URL of the online materials for their experimental condition.

Procedure for the Experiment

1) Sign the consent form if 18 years of age or older and the assent form if under 18. Put the 6-digit code from the scratch paper on your form. Be sure to use your legal name at Georgia Tech on the form.
2) Put all of your items under the table. Please do not have your phone on the table! Please only use the exam materials and do not go to any other website during the study.
3) Fill out the demographic survey at https://www.surveymonkey.com/r/Ex3Demo use your 6-digit code at the beginning of the survey.
4) Go to http://tinyurl.com/Ex3-reg and register. **Use your 6-digit code as your username**. Use **thinkcspy** for the course name. You only have to enter the course name once, not twice. Record your 6-digit code and password for use next week.
5) Go to the url on the paper you were given and complete the following, one after the other. Click on the first link in the table of contents to start.

    a. Practice Material – introduction and practice for each type of exam: Multiple choice (2), Fix code (1), Order code (1), and Write code (1)
    b. Pretest – **hand in your pretest scratch paper after you complete this at the front of the room**. It has Multiple choice (5), Fix code (1), Order code (1) and Write code (1)
    c. Instructional Material - four pairs of a correct example and a similar practice problem. Try to solve each practice problem (4).
    d. Posttest – **hand in your posttest scratch paper after you complete this as you leave.** Please come back next week at 3pm to take the 2nd posttest. It should only take 30 minutes to 1 hour.

If you have any trouble raise your hand or come up to the front of the room. Thank you for taking part in the experiment!

Click on the "Start" button 

to start each timed exam. Be sure to click the "Finish Exam" button 

when you are done with each timed exam.

If you see the **"Are you sure you want to leave this page"** alert, click **"Stay on page"** and be sure to click the **"Finish Exam"** button.

Only use the right arrow button to get to the next page. Do not go back to the table of contents! Please only use the materials for the study. Do not open any other websites!

If you have to go to the bathroom during the study you can, but do not take anything with you.

There are many free interactive ebooks at http://runestoneinteractive.org/library.html that you may find helpful.

Student Demographic Survey

The student demographic survey was the same as from the first study on the effectiveness and efficiency of Parsons problems versus fix code and write code from chapter four.

REFERENCES

Alvarado, C., Morrison, B., Ericson, B., Guzdial, M., Miller, B., & Ranum, D. (2012). *Performance and use evaluation of an electronic book for introductory Python programming*. Retrieved from Georgia Institute of Technology: http://hdl.handle/1853/45044

Anderson, J. R. (1983). *The Architecture of Cognition*: Harvard University Press.

Anderson, J. R., Bothell, D., & Byrne, M. D. (2004). An Integrated Theory of the Mind. *Psychological Review, 111*(4), 1036-1060.

Anderson, J. R., Boyle, C. F., Corbett, A. T., & Lewis, M. W. (1990). Cognitive modeling and intelligent tutoring *Artificial intelligence and learning environments* (pp. 7-49): Bradford Company.

Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive Tutors: Lessons Learned. *Journal of the Learning Sciences, 4*(2), 167-207. doi:10.1207/s15327809jls0402_2

Astrachan, O., Cuny, J., Stephenson, C., & Wilson, C. (2011). *The CS10K Project: Mobilizing the Community to Transform High School Computing*. Paper presented at the Proceedings of the 42nd ACM technical symposium on Computer science education., Dallas Texas, USA.

Atkinson, R. K., Derry, S. J., Renkl, A., & Wortham, D. (2000). Learning from Examples: Instructional Principles from the Worked Examples Research. *Review of Educational Research, 70*(2), 181–214.

Baecker, R., & Small, I. (1990). Animation at the interface. *The art of human-computer interface design*, 251-267.

Barnes, T., & Stamper, J. (2008). *Toward automatic hint generation for logic proof tutoring using historical student data.* Paper presented at the International Conference on Intelligent Tutoring Systems.

Benda, K., Bruckman, A., & Guzdial, M. (2012). When Life and Learning Do Not Fit: Challenges of Workload and Communication in Introductory Computer Science Online. *Trans. Comput. Educ., 12*(4), 1-38. doi:10.1145/2382564.2382567

Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *SIGCSE Bull., 39*(2), 32-36. doi:10.1145/1272848.1272879

Berk, L. E., & Winsler, A. (1995). *Scaffolding Children's Learning: Vygotsky and Early Childhood Education*: National Association for the Education of Young Children.

Bjork, E. L., & Bjork, R. A. (2011). Making things hard on yourself, but in a good way: Creating desirable difficulties to enhance learning. *Psychology and the real world: Essays illustrating fundamental contributions to society*, 56-64.

Bjork, R. A., Dunlosky, J., & Kornell, N. (2013). Self-Regulated Learning: Beliefs, Techniques, and Illusions. *Annual Review of Psychology, 64*, 417-444.

Bloom, B. S. (1984). The 2 Sigma Problem: The Search for Methods of Group Instruction as Effective as One-to-One Tutoring. *Educational Researcher, 13*(6), 4-16. doi:10.2307/1175554

Boulay, B. D. (1988). Some Difficulties of Learning to Program. In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer* (pp. 283-299): Lawrence Erlbaum Associates.

Bransford, J. D., Brown, A. L., & Cocking, R. R. (Eds.). (2000). *How People Learn*. Washington D.C.: NATIONAL ACADEMY PRESS.

Bruckman, A., Biggers, M., Ericson, B., McKlin, T., Dimond, J., DiSalvo, B., . . . Yardi, S. (2009). *"Georgia computes!": improving the computing education pipeline*. Paper presented at the Proceedings of the 40th ACM technical symposium on Computer science education, Chattanooga, TN, USA.

Chandler, P., & Sweller, J. (1992). The split-attention effect as a factor in the design of instruction. *British Journal of Educational Psychology, 62*(2), 233–246.

Chaudhri, V. K., Cheng, B., Overtholtzer, A., Roschelle, J., Spaulding, A., Clark, P., . . . Gunning, D. (2013). Inquire Biology: A textbook that answers questions. *AI Magazine, 34*(3), 55-72.

Cheng, N., & Harrington, B. (2017). *The Code Mangler: Evaluating Coding Ability Without Writing any Code*. Paper presented at the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, Seattle, Washington, USA.

Chevalier, F., Riche, N. H., Plaisant, C., Chalbi, A., & Hurter, C. (2016). *Animations 25 Years Later: New Roles and Opportunities*. Paper presented at the Proceedings of the International Working Conference on Advanced Visual Interfaces, Bari, Italy.

Chi, M. T. H. (2009). Active-Constructive-Interactive: A Conceptual Framework for Differentiating Learning Activities. *Topics in Cognitive Science, 1*, 73-105. doi:10.1111/j.1756-8765.2008.01005.x

Chi, M. T. H., Bassok, M., Lewis, M., Reimann, P., & Glaser, R. (1989). Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science, 13*, 145-182.

Clark, R. C., & Mayer, R. E. (2011). *E-Learning and the Science of Instruction: Proven Guidelines for Consumers and Designers of Multimedia Learning* Pfeiffer.

Cooper, G., & Sweller, J. (1987). The effects of schema acquisition and rule automation on mathematical transfer. *Journal Educational* 347-362.

Corbalan, G., Kester, L., & Merrieonbeor, J. J. G. v. (2008). Selecting learning tasks: Effects of adaptations and shared control on learning efficiency and task involvement. . *Contemporary Educational Psychology, 33*, 733-756.

Corbett, A. T., Koedinger, K. R., & Anderson, J. R. (1997). Intelligent tutoring systems. *Handbook of human-computer interaction, 5*, 849-874.

Crouch, C. H., & Mazur, E. (2001). Peer instruction: Ten years of experience and results. *American Journal of Physics, 69*(9), 970-977.

Cunningham, K., Blanchard, S., Ericson, B., & Guzdial, M. (2017). *Using Tracing and Sketching to Solve Programming Problems: Replicating and Extending an Analysis of What Students Draw*. Paper presented at the 2017 ACM Conference on International Computing Education Research, Tacoma, Washington, USA.

Denny, P., Luxton-Reilly, A., & Simon, B. (2008). *Evaluating a New Exam Question: Parsons Problems*. Paper presented at the International Computing Education Research Conference, Sydney, Australia.

Dewey, J. (1959). *Experience & Education*. New York: Macmillan.

Druckman, D., & Bjork, R. A. (1991). *In the mind's eye: Enhancing human performance*. Washington DC: National Academy Press.

Dweck, C. (1986). Motivational Processes Affecting Learning. *American Psychologist, 41*(10), 1040-1048.

Ericson, B., Engelman, S., McKlin, T., & Taylor, J. Q. (2014). *Project rise up 4 CS: increasing the number of black students who pass advanced placement CS A*. Paper presented at the Proceedings of the 45th ACM technical symposium on Computer science education, Atlanta, Georgia, USA. http://dl.acm.org/citation.cfm?doid=2538862.2538937

Ericson, B., Guzdial, M., & Biggers, M. (2005). *A model for improving secondary CS education*. Paper presented at the Proceedings of the 36th SIGCSE technical symposium on Computer science education, St. Louis, Missouri, USA.

Ericson, B., Guzdial, M., Morrison, B., Parker, M., Moldavan, M., & Surasani, L. (2015). An eBook for teachers learning CS principles. *ACM Inroads, 6*(4), 84-86.

Ericson, B., & McKlin, T. (2015). *Helping African American Students Pass Advanced Placement Computer Science: A Tale of Two States*. Paper presented at the Research on Equity and Sustained Participation in Engineering, Computing, and Technology (RESPECT), Charlotte, NC, USA.

Ericson, B., Moore, S., Morrison, B., & Guzdial, M. (2015). *Usability and Usage of Interactive Features in an Online Ebook for CS Teachers*. Paper presented at the Proceedings of the Workshop in Primary and Secondary Computing Education, London, United Kingdom. http://dl.acm.org/citation.cfm?doid=2818314.2818335

Ericson, B., Rogers, K., Parker, M., Morrison, B., & Guzdial, M. (2016). *Identifying Design Principles for CS Teacher Ebooks through Design-Based Research*. Paper presented at the 2016 ACM Conference on International Computing Education Research, Melbourne, VIC, Australia.

Ericson, B. J., Guzdial, M. J., & Morrison, B. B. (2015). *Analysis of Interactive Features Designed to Enhance Learning in an Ebook*. Paper presented at the 2015 ACM Conference on International Computing Education Research, Omaha, NE, USA.

Ericson, B. J., Parker, M. C., & Engelman, S. (2016). *Sisters Rise Up 4 CS: Helping Female Students Pass the Advanced Placement Computer Science A Exam*. Paper presented at the Proceedings of the 47th ACM Technical Symposium on Computing Science Education, Memphis, Tennessee, USA.

Ericson, B. J., Rick, J., & Margulieux, L. E. (2017). *Solving Parsons Problems Versus Fixing and Writing Code*. Paper presented at the 17th Koli Calling International Conference on Computing Education Research, Koli, Finland.

Ericsson, K. A. (2006). The Influence of Experience and Deliberate Practice on the Development of Superior Expert Performance *The Cambridge Handbook of Expertise and Expert Performance*. New York: Cambridge University Press.

Ericsson, K. A., Krampe, R. T., & Tesch-Romer, C. (1993). The role of deliberate practice in the acquisition of expert performance. *Psychological Review, 100*(3), 363-406.

Fisler, K. (2014). *The recurring rainfall problem*. Paper presented at the Proceedings of the tenth annual conference on International computing education research, Glasgow, Scotland, United Kingdom.

Folsom-Kovarik, J. T., Schatz, S., & Nicholson, D. (2010). *Plan ahead: Pricing ITS learner models*. Paper presented at the Proceedings of the 19th Behavior Representation in Modeling & Simulation (BRIMS) Conference.

Garner, S. (2007). An Exploration of How a Technology-Facilitated Part-Complete Solution Method Supports the Learning of Computer Programming. *Journal of Issues in Informing Science and Information Technology, 4*, 491-501.

Goel, A. K., & Polepeddi, L. (2018). Jill Watson: A Virtual Teaching Assistant for Online Education. In C. Dede, J. Richards, & B. Saxberg (Eds.), *Education at scale: Engineering online teaching and learning*. New York: Routledge.

Gravetter, F. J., & Wallnau, L. B. (2016). *Statistics for the behavioral sciences*: Cengage Learning.

Greeno, J. G., Collins, A. M., & Resnick, L. B. (1996). Cognition and Learning. In D. C. Berliner & R. C. Calfee (Eds.), *Handbook of Educational Psychology*. New York: Macmillan.

Harms, K. J., Chen, J., & Kelleher, C. L. (2016). *Distractors in Parsons Problems Decrease Learning Efficiency for Young Novice Programmers*. Paper presented at the 2016 ACM Conference on International Computing Education Research, Melbourne, VIC, Australia.

Harms, K. J., Rowlett, N., & Kelleher, C. (2015). *Enabling Independent Learning of Programming Concepts through Programming Completion Puzzles*. Paper presented at the Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Atlanta, GA.

Harns, K. (2015). *The Impact of Distractors in Programming Completion Puzzles on Novice Programmers Position Statement*. Paper presented at the Blocks and Beyond Workshop, Atlanta, GA.

Helminen, J., Ihantola, P., Karavirta, V., & Alaoutinen, S. (2013). *How Do Students Solve Parsons Programming Problems? – Execution-based vs. line-based feedback*. Paper presented at the Proceedings of Learning and Teaching in Computing and Engineering (LaTiCE), Macau.

Helminen, J., Ihantola, P., Karavirta, V., & Malmi, L. (2012). *How Do Students Solve Parsons Programming Problems? - An Analysis of Ineraction Traces*. Paper presented at the International Computing Education Research Conference, Aukland, New Zealand.

Hmelo, C. E., & Guzdial, M. (1996). *Of black and glass boxes: scaffolding for doing and learning*. Paper presented at the Proceedings of the 1996 international conference on Learning sciences, Evanston, Illinois.

Honig, W. L. (2013). *Teaching and assessing programming fundamentals for non majors with visual programming*. Paper presented at the Proceedings of the 18th ACM conference on Innovation and technology in computer science education, Canterbury, England, UK.

Horwitz, S., Rodger, S. H., Biggers, M., Binkley, D., Frantz, C. K., Gundermann, D., . . . Sweat, M. (2009). *Using peer-led team learning to increase participation and success of under-represented groups in introductory computer science*. Paper presented at the Proceedings of the 40th ACM technical symposium on Computer science education, Chattanooga, TN, USA.

Hutchins, E. (1995). How a Cockpit Remembers Its Speed. *Cognitive Science, 19*, 265-288.

Ihantola, P., Helminen, J., & Karavirta, V. (2013). *How to study programming on mobile touch devices: interactive Python code exercises*. Paper presented at the Proceedings of the 13th Koli Calling International Conference on Computing Education Research, Koli, Finland. http://dl.acm.org/citation.cfm?doid=2526968.2526974

Ihantola, P., & Karavirta, V. (2011). Two-Dimensional Parson's Puzzles: The Concept, Tools, and First Observations. *Journal of Information Technology Education, 10*, 1-14.

Jin, W., Barnes, T., Stamper, J., Eagle, M., Johnson, M., & Lehmann, L. (2012). *Program representation for automatic hint generation for a data-driven novice programming tutor.* Paper presented at the Intelligent Tutoring Systems.

Kalyuga, S. (2007). Expertise reversal effect and its implications for learner-tailored instruction. *Educational Psychology Review, 19*, 509-539.

Kalyuga, S., Ayres, P., Chandler, P., & Sweller, J. (2003). The expertise reversal effect. *Educational Psychologist, 38*(1), 23-31.

Karavirta, V., Helminen, J., & Ihantola, P. (2012). *A mobile learning application for parsons problems with automatic feedback*. Paper presented at the Proceedings of the 12th Koli Calling International Conference on Computing Education Research, Koli, Finland.

Kelleher, C., Pausch, R., & Kiesler, S. (2007). *Storytelling alice motivates middle school girls to learn computer programming*. Paper presented at the Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, San Jose, California, USA.

Kinnunen, P., & Simon, B. (2010). *Experiencing programming assignments in CS1: the emotional toll*. Paper presented at the Proceedings of the Sixth international workshop on Computing education research, Aarhus, Denmark. http://dl.acm.org/citation.cfm?doid=1839594.1839609

Knight, J. K., & Wood, W. B. (2005). Teaching more by lecturing less. *Cell biology education, 4*(4), 298-310.

Kober, N. (2015). *Reaching Students: What Research Says about Effective Instruction in Undergraduate Science and Engineering*: ERIC.

Korhonen, A., Naps, T., Boisvert, C., Crescenzi, P., Karavirta, V., Mannila, L., . . . Shaffer, C. A. (2013). *Requirements and design strategies for open source interactive computer science eBooks*. Paper presented at the ITiCSE working group reports conference on Innovation and technology in computer science education-working group reports, Canterbury, England, United Kingdom. http://dl.acm.org/citation.cfm?doid=2543882.2543886

Lave, J., & Wenger, E. (1991). *Situated learning: Legitimate peripheral participation*: Cambridge University Press.

Le, N.-T., Strickroth, S., Gross, S., & Pinkwart, N. (2013). A review of AI-supported tutoring approaches for learning programming *Advanced Computational Methods for Knowledge Engineering* (pp. 267-279): Springer.

LeFevre, J. A., & Dixon, P. (1986). Do written instructions need examples? *Cognition and Instruction, 3*, 1-30.

Leppink, J., Paas, F., Vleuten, C. P. M. V. d., Gog, T. V., & Merriënboer, J. J. G. V. (2013). Development of an instrument for measuring different types of cognitive load. *Behavior research methods, 45*(4), 1058-1072.

Liao, S. N., Zingaro, D., Laurenzano, M. A., Griswold, W. G., & Porter, L. (2016). *Lightweight, Early Identification of At-Risk CS1 Students*. Paper presented at the Proceedings of the 2016 ACM Conference on International Computing Education Research, Melbourne, VIC, Australia.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hammer, J., Lindholm, M., . . . Thomas, L. (2004). *A Multi-National Study of Reading and Tracing Skills in Novice Programmers*. Paper presented at the Working group reports from ITiCSE on Innovation and technology in computer science education, Leeds, United Kingdom.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch Programming Language and Environment. *Trans. Comput. Educ., 10*(4), 1-15. doi:10.1145/1868358.1868363

Margolis, J., & Fisher, A. (2003). *Unlocking the Clubhouse: Women in Computing*. Cambridge, MA: The MIT Press.

Mayer, R. E. (2005). Principles for managing essential processing in multimedia learning: Segmenting, pretraining, and modality principles. *The Cambridge handbook of multimedia learning* (pp. 147-158). New York: Cambridge University Press.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., . . . Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bull., 33*(4), 125-180. doi:10.1145/572139.572181

Merriënboer, J. J. G. V. (1990). Strategies for programming instruction in high school: Program completion vs. program generation. *Journal of Educational Computing Research, 6*(3), 265-285.

Merriënboer, J. J. G. V., & Croock, M. B. M. D. (1992). Strategies for computer-based programming instruction: program completion vs. program generation. *Journal of Educational Computing Research, 8*(3), 365-394.

Merriënboer, v., Kirschner, & Kester. (2003). Taking the Load Off a Learner's Mind: Instructional Design for Complex Learning. *Educational Psychologist, 38*(1), 5-13.

Miller, B., & Ranum, D. (2013). *How to Think Like a Computer Scientist Learning with Python: Interactive Edition* (2.0 ed.).

Miller, B. N., & Ranum, D. L. (2012). Beyond PDF and ePub: toward an interactive textbook *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education %@ 978-1-4503-1246-2* (pp. 150-155). Haifa, Israel: ACM.

Morrison, B. B., Dorn, B., & Guzdial, M. (2014). *Measuring cognitive load in introductory CS: adaptation of an instrument*. Paper presented at the tenth annual conference on International computing education research, Glasgow, Scotland, United Kingdom.

Morrison, B. B., Margulieux, L. E., Ericson, B., & Guzdial, M. (2016). *Subgoals Help Students Solve Parsons Problems*. Paper presented at the 43rd ACM technical symposium on Computer Science Education, Memphis, Tennessee.

Norman, D. A. (2002). *The Design of Everyday Things*: Basic Books, Inc.

Papert, S. (1980). *Mindstorms: children, computers, and powerful ideas*: Basic Books, Inc.

Parsons, D., & Haden, P. (2006). *Parson's programming puzzles: a fun and effective learning tool for first programming courses*. Paper presented at the 8th Australasian Conference on Computing Education, Hobart, Australia.

Pausch, R. (2008). *Alice: a dying man's passion*. Paper presented at the Proceedings of the 39th SIGCSE technical symposium on Computer science education, Portland, OR, USA.

Pirolli, P. L., & Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology, 39*(2), 240-272.

Porter, L., Bouvier, D., Cutts, Q., Grissom, S., Lee, C., McCartney, R., . . . Simon, B. (2016). *A Multi-institutional Study of Peer Instruction in Introductory Computing*. Paper presented at the Proceedings of the 47th ACM Technical Symposium on Computing Science Education, Memphis, Tennessee, USA.

Porter, L., Lee, C. B., Simon, B., & Zingaro, D. (2011). *Peer instruction: do students really learn from peer discussion in computing?* Paper presented at the Proceedings of the seventh international workshop on Computing education research, Providence, Rhode Island, USA. http://dl.acm.org/citation.cfm?doid=2016911.2016923

Porter, L., Zingaro, D., & Lister, R. (2014). *Predicting student success using fine grain clicker data*. Paper presented at the Proceedings of the tenth annual conference on International computing education research, Glasgow, Scotland, United Kingdom.

Rivers, K., & Koedinger, K. R. (2017). Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education, 27*(1), 37-64.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education, 13*(2), 137-172.

Rogoff, B. (1990). Apprenticeship in thinking: Cognitive development in sociocultural activity: New York: Oxford University Press.

Rohrer, D., & Taylor, K. (2006). The effects of over-learning and distributed practice on the retention of mathematics knowledge. *Applied Cognitive Psychology, 20*, 1209-1224.

Rosenbaum, D. A., Carlson, R. A., & Gimore, R. O. (2001). Acquisition of intellectual and perceptual motor skills. *Annual Review of Psychology, 52*, 453-470.

Seaton, D. T., Bergner, Y., Chuang, I., Mitros, P., & Pritchard, D. E. (2014). Who does what in a massive open online course? *Commun. ACM, 57*(4), 58-65. doi:10.1145/2500876

Shaffer, D. W., & Resnick, M. (1999). "Thick" Authenticity: New Media and Authentic Learning. *Journal of Interactive Learning Research, 10*(2), 195-215.

Simon. (2013). *Soloway's Rainfall Problem has become Harder*. Paper presented at the 2013 Learning and Teaching in Computing and Engineering.

Simon, H. A., & Chase, W. G. (1973). *Skill in chess* (Vol. 61): American Scientist.

Slobada, J. A., Davidson, J. W., Howe, M. J. A., & Moore, D. G. (1996). The role of practice in the development of performing musicians. *British Journal of Educational Psychology, 87*, 287-309.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM, 29*(9), 850-858.

Soloway, E., Guzdial, M., & Hay, K. E. (1994). Learner-Centered Design: The Challenge For HCI In The 21st Century. *Interactions, 1*(2), 36-48.

Spohrer, J. C., & Soloway, E. (1986). Novice Mistakes: are the folk wisdoms correct? *Communications of the ACM, 29*(7), 624-632.

Sweller, J. (2004). Instructional design consequences of an analogy between evolution by natural selection and human cognitive architectures. *Instructional Science, 32*, 9-31.

Sweller, J. (2010). Cognitive Load Theory: Recent Theoretical Advances. In J. L. Plass, R. Moreno, & R. Brünken (Eds.), *Cognitive Load Theory*: Cambridge University Press.

Sweller, J., & Cooper, G. (1985). The Use of Worked Examples as a Substitute for Problem Solving in Learning Algebra. *Cognition and Instruction, 2*(1), 59-89.

Tew, A. E., & Guzdial, M. (2010). *Developing a validated assessment of fundamental CS1 concepts*. Paper presented at the 41st ACM technical symposium on Computer science education, Milwaukee, Wisconsin, USA. http://dl.acm.org/citation.cfm?doid=1734263.1734297

Trafton, J. G., & Reiser, B. J. (1993). *The contributions of studying examples and solving problems to skill acquisition*. Paper presented at the 15th Annual Conference of the Cognitive Science Society, Hillsdale, NJ. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.9933&rep=rep1&type=pdf

Trochim, W. M., & Donnelly, J. P. (2006). *Research methods knowledge base (3rd ed)*. Cincinnati, OH: Atomic Dog.

Tuffiash, M., Roring, R. W., & Ericsson, K. A. (2007). Expert performance in Scrabble: Implications for the study of the structure and acquisition of complex skills. *Journal of Experimental Psychology: Applied, 13*, 124-134.

Utting, I., Tew, A. E., McCracken, M., Thomas, L., Bouvier, D., Frye, R., . . . Wilusz, T. (2013). *A fresh look at novice programmers' performance and their teachers' expectations*. Paper presented at the ITiCSE working group reports conference on Innovation and technology in computer science education-working group reports Canterbury, England, United Kingdom. http://dl.acm.org/citation.cfm?doid=2543882.2543884

Vanlehn, K. (2006). The behavior of tutoring systems. *International Journal of Artificial Intelligence in Education, 16*(3), 227-265.

Vygotsky, L. S. (1978). *Mind in society: The development of higher mental processes*. Cambridge, MA: Harvard University Press.

Wadsworth, B. J. (1989). *Piaget's Theory of Cognitive and Affective Development - Fourth Edition*. New York Longman.

Ward, M., & Sweller, J. (1990). Structuring Effective Worked Examples. *Cognition and Instruction, 7*(1), 1–39.

Ward, P., Hodges, N. J., & Starkes, J. L. (2004). Deliberate practice and expert performance: Defining the path to excellence. In A. M. Williams & N. J. Hodges (Eds.), *Skill acquisition in sport: Research, theory and practice*. London, UK: Routhledge.

Watson, C., & Li, F. W. B. (2014). *Failure rates in introductory programming revisited*. Paper presented at the 2014 conference on Innovation and technology in computer science education, Uppsala, Sweden.

Winslow, L. E. (1996). Programming Pedagogy - A Psychological Overview. *SIGCSE Bull., 28*(3), 17-22. doi:10.1145/234867.234872

Zhu, X., & Simon, H. A. (1987). Learning mathematics from examples and by doing. *Cognition and Instruction, 4*, 137-166

.